THE RECONFIGURABLE MULTIRING AND APPLICATIONS

by

GITA C. WILLIAMS

(Under the Direction of Hamid R. Arabnia)

ABSTRACT

The goal of this dissertation was to further study the MultiRing network which was first introduced by Arabnia [8]. Novel communication models have been developed that can be used at the high-level to develop programs specifically for use on the MultiRing or at the low-level for routing messages. These models include the pipeline, cube and tree communication models. Existing algorithms developed for various well-known static interconnection networks can be mapped onto the MultiRing using these models.

New algorithms for routing operations, namely ring broadcasting, group broadcasting and distributing have be created for use on the MultiRing. These algorithms are based on enabling multiprogramming where multiple programs running together on the same network use disjoint subsets of the MultiRing processors.

A novel switch design is introduced which makes it possible for to expand the MultiRing to allow additional processors with minimal cost . It allows existing ring formations to remain intact when expanding the network. This new scaleable digital layout features grouping processors in pairs and building larger switches for modular switching elements.

A MultiRing simulator has been created on which programmers can create and test parallel algorithms. All of the MultiRing routines are stored in a header file which must be included in C program. Programmers have access to over 50 functions to use on the MultiRing.

The motivation behind this research work was to study various properties of the MultiRing and incorporate concepts of multiprogramming in all aspects of the design-- from the construction of the switch to the communication strategies used between processors.

INDEX WORDS: Parallel Algorithms, Parallel Computer, Reconfigurable Architecture, Multiprogramming, Interconnection Networks

THE RECONFIGURABLE MULTIRING AND APPLICATIONS

by

GITA C. WILLIAMS

B.S., Georgia College, 1990

M.S., The University of Georgia, 1992

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2003

© 2003

Gita C. Williams

All Rights Reserved

THE RECONFIGURABLE MULTIRING AND APPLICATIONS

by

GITA C. WILLIAMS

Approved:

Major Professor: Hamid R. Arabnia

Committee: Jay E. Aronson
Suchendra M. Bhandarkar
Robert M. Branch
Thiab R. Taha

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2003

DEDICATION

This dissertation is dedicated to my mother, Mrs. Charlia E. Williams, a retired educator who taught me to always do the very best I can and to never give up. Thank you for listening to my tears, making me confront my fears and encouraging me daily through laughter. I am truly blessed to have you as my mother.

ACKNOWLEDGEMENTS

I am so very grateful to my advisor, Professor Hamid Arabnia, for constantly encouraging me and believing in me. I appreciate your leadership and expertise. It has been such a learning experience to work under your tutelage. I will always remember the Monday mornings you cleared your schedule to ensure I had time to discuss the various stages of my research. Like you, I hope to make students a priority in my life. Thank you for encouraging me to pursue this research topic and keeping me on the right path to completing this degree.

I extend my heartfelt appreciation for the guidance from my dissertation committee. I am very grateful to Professors Suchendra M. Bhandarkar and Thiab R. Taha for taking the time to meet with me. I value the suggestions you made to help me improve this dissertation. Professor Robert Branch, your encouraging words over the years have kept me grounded. I learned so much in your Instructional Technology courses and I have already taken the lessons you taught into my own classroom. Professor Jay Aronson, during my dissertation planning and prospectus meeting, your support and encouraging words helped to calm my fears (and tears☺). Professor Arabnia has told me he is very grateful for all of the support from each of you. This dissertation would not have been possible without all of you. Thank you.

It has taken me 7 ½ year to complete this dissertation; I never knew it would be so difficult to work and to go to school at the same time. I give thanks to all of my "chairs" for their support throughout the years: Professors David Devries, John S. Robertson, Craig Turner, Gerald Adkins and Lila Roberts from Georgia College & State University and Professors Robert

Robinson, Rodney Canfield and Krys Kochut from the University of Georgia. I appreciate the long hours you spent discussing algorithms and proofs, meeting me early Saturday mornings to discuss mathematical formulas, providing financial support, rearranging my teaching schedule to allow me to live in Athens to concentrate on finishing this PhD, loaning me your laptop, letting me have an office near Professor Arabnia and encouraging me to continue and finish.

Thank you, Professor Jeffery Smith for taking time out of your busy schedule to discuss circuit design with me and Professor Don Potter for donating without hesitation computers to use for my "lab." Thank you Professors Miller, He, Valafar and Deligiannidis, Jill, Novene, Claudia, Jean, Elizabeth, Nathan, Ken and Piotr for your help. Special thanks are given to my mentor, Professor Mae Carpenter, who emailed me just when I needed extra encouragement and Professor Lucretia Coleman for constantly checking to make sure I was OK.

To my true friends that have supported me through my tears, Chandra Williams, Professor Rajade Berry-James, Sylvia Jones and Mimi Kendrick: I am truly blessed to have you in my life.

This entire process has shown me how valuable my family is. Thank you mommy, Dixie (my dog), Mygleetus, Colette, Phil, Cody, Kenneth and Curtis.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1 : INTRODUCTION

Many applications require greater computational speed and memory capacity than what is possible on a single computer. The answer to this is *parallel computing*, which involves having more than one processor (computer) working simultaneously on different parts of the same problem. Ideally, parallelism can be used to reduce processing time and increase data workload for both scientific and commercial applications.

Notably, the U.S. High Performance Computing and Communications program has identified several Grand Challenge applications in science and engineering whose solutions can be advanced with parallel processing [33] One category of Grand Challenge problems being investigated by Computational Technologies for Earth and Space Sciences involves monitoring global change and earth modeling [27] A few computationally intensive research problems in this category include:

- Integration of major U.S. climate and numerical weather predictions
- Atmospheric and oceanic data assimilation systems
- Multi-year Southern California earthquake forecast
- Modeling regional and global terrestrial water and energy cycles
- Real-time space weather prediction capability using solar and interplanetary observations

The desire for visualization of the results adds to the demand for faster processing.

The need for parallelism is great in the commercial world, too, since the amount of work a business can handle is determined by its computer system speed and capacity. As a business

grows, its customer database grows proportionally, and memory capacity demands increase. A fast system is needed to generate results quickly and accurately.

It would be best to have a single very fast and very accurate processor for science and businesses to use; but the demands for computational performance continue to outpace what an individual processor can deliver. Furthermore, there is a limit (not yet reached) to the speed of a single processor, which is the speed of light [26] Computer scientists serving business, science and governments can wait for a faster processor to meet their needs or they can construct parallel systems using a set of current state-of-the-art processors. If they are assembled from commercially available module sets and have a limited number of customizations, parallel systems can be the cheaper and better alternative [3].

Today, the same processor technology is used throughout the range of available computer systems, from low-end individual desktop to high-end supercomputers. Increasing the *number* of processors can result in a system with performance greater than what the state-of-the-art processor is capable of. According to Culler, Sing and Gupta [26] parallel computer architectures "translate the raw potential of technology into greater performance and expanded system capability."

1.1 The Need for the MultiRing Interconnection Network

The multiple processors in a parallel system must be connected in order to communicate and share information. In fact, the major overhead in parallelism is from communication costs. How the interconnection network is designed affects the performance and scalability of a parallel computer system and the complexity of parallel programs. Various ways exist to connect processors for inter-processor communication, including static interconnection networks that cannot change during a program's execution and dynamic networks that can be reconfigured

while a program executes. Static networks work best with applications that have a predictable communication pattern matching the topology of the network. Dynamic networks, since they support a variety of communication patterns, are considered general purpose networks. Chapter 2 provides an overview of static and dynamic networks.

To provide a scaleable interconnection network with efficient processor communication requires a balance between the number of physical links in the network and the length of the communication path connecting two processors. Efficient processor communication requires a short path between processors; optimally there should be a direct link connecting two communicating processors. On the other hand, a scaleable computer system involves using a small number of physical links to connect processors. The MultiRing is a dynamic interconnection network that provides both efficient processor communication and a limited number of links.

Many computer labs today are networked in a static ring network in which processors communicate by sending messages to each other. A ring is scaleable: because each processor has only two links to connect to other processors, it is easy to add additional processors in an existing ring. However, the communication path (network diameter) is long because a message circulates around the ring of processors until it reaches its destination. When the MultiRing is employed it reduces the length of the communication path of a ring. It does this by introducing alternate paths that can bring a message closer to its destination. Thus, applications running on a MultiRing will potentially execute faster than applications on a simple ring network because communication overhead is reduced.

Instead of connecting processors in a single static ring, the MultiRing dynamically forms multiple rings of processors. The network can be configured to form many small rings or a few

large rings of processors while programs are executing. The strict requirements for establishing the rings lead to alternate paths for transmitting messages. As the program executes, the connections between processors are reconfigured to meet the required communication needs. By waiting for a specific configuration of the network before transmitting messages, processors decrease the number of intermediate processors a message must pass through before reaching its destination. This increases speed.

The MultiRing can be used in a massively parallel environment with thousands of processors since it is scaleable. Each processor uses only two links to connect to a MultiRing Switch composed of inexpensive modular switching elements. Chapter 3 presents the digital design of the switching elements and provides a detailed description of how the MultiRing Switch expands to accommodate additional processors.

Existing parallel applications designed for static networks can be implemented on the MultiRing. In particular, Arabnia and Bhandarkar have theorized that algorithms designed for 2D grids and hypercubes can be simulated on the MultiRing [8, 17]. The research being presented here has taken the theories one step further. Chapter 4 presents practical details of simulating pipelines, n-cubes, 2D grids and complete binary trees on a MultiRing provided only a subset of links are used at a given time.

The multistage interconnection MultiRing network also differs from other dynamic networks in its approach to handling collisions. A collision occurs when two messages reach the same processor simultaneously, resulting in the possibility of one message being lost. The MultiRing avoids collisions by limiting the number of other processors one processor can directly connect to at a given time. With strict reconfiguration guidelines, a processor is

connected to, at most, two processors at a time. A communication-assist server on each processor can be used to direct the flow of data on the network.

1.2 Brief Description of the MultiRing

A *node* is a parallel processing term that can refer to one or more processors; however, in our illustrations, for simplicity, a node contains a single processor. A MultiRing is a dynamic interconnection network in which each node has two physical links that connect to a central MultiRing Switch. The switch has a limited number of configurations in which nodes are connected to form multiple independent rings; hence the name "MultiRing." In Figure 1.1, 8 nodes labeled $P_0 - P_7$ connect to the MultiRing Switch.

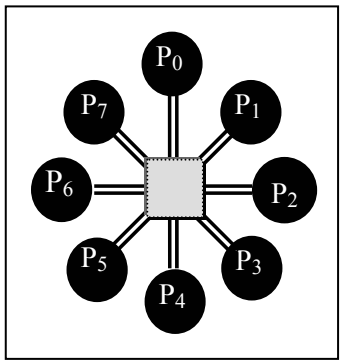


Figure 1.1: Nodes use 2 links to connect to the MultiRing Switch.

Nodes communicate with each other through the MultiRing Switch. The MultiRing Switch forms balanced rings of nodes in which all rings have the same number of nodes. In a network of $N=2^r$ nodes, there are $(r + 1)$ different configurations with 2^i rings of 2^{r-i+1} nodes where $0 \leq i < r$. For example, an 8-node MultiRing can be reconfigured 4 ways, 1 ring of 8 nodes, 2 rings of 4 nodes, 4 rings of 2 nodes and 8 rings of 1 node. There are different ways to

illustrate the configurations. Figure 1.2 emphasizes that each node has two physical links connecting to a switch box which internally form multiple rings. Figure 1.3 emphasizes the direct connections available on each configuration of an 8-node MultiRing.

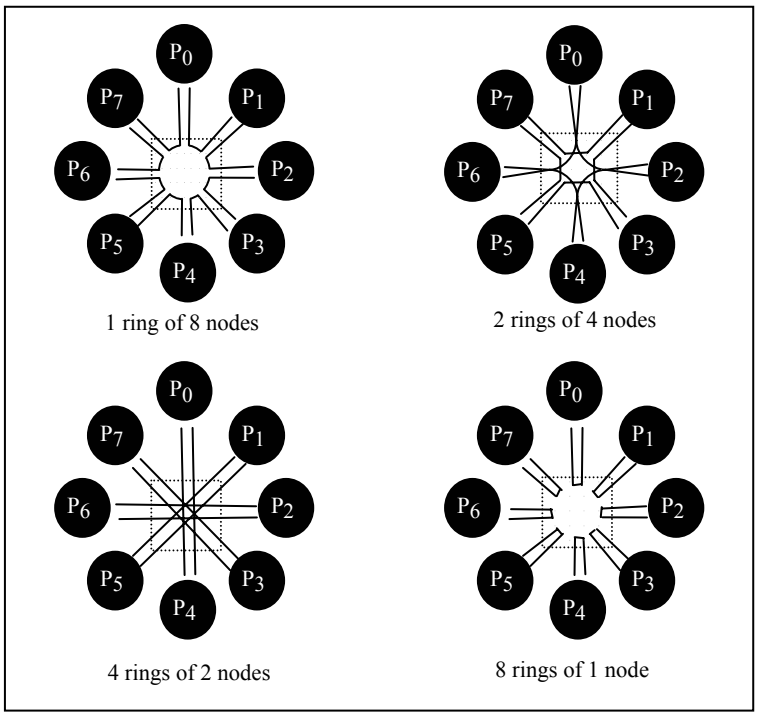


Figure 1.2: Configurations of an 8-node MultiRing Switch

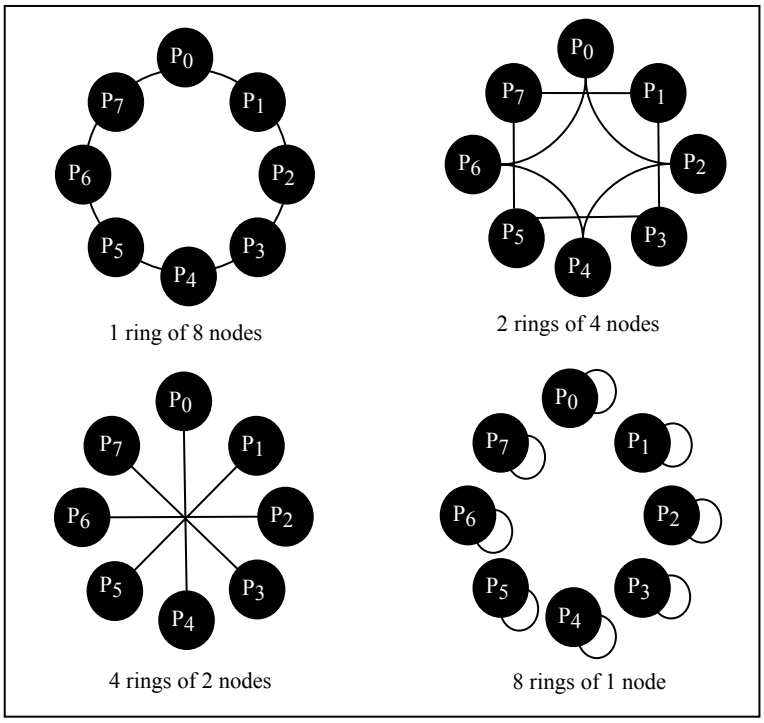


Figure 1.3: Direct connections in an 8-node MultiRing

The switch is limited in that it does not allow each node to directly connect with all other nodes. For scalability, only a subset of all possible direct connections is allowed. In a configuration of an N-node MultiRing with 2^i rings, a node connects to nodes physically 2^i units away where $N = 2^r$ and $0 \leq i < r$. In total, a node directly connects to $2r$ different nodes in turn. Figure 1.4 shows the six direct connections from node P_0 in an 8-node MultiRing.

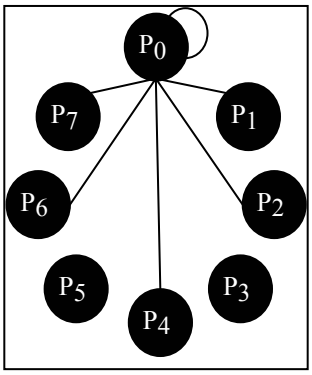


Figure 1.4 : Direct connections for P_0

Nodes that are not directly connected can still communicate efficiently. Messages can be routed through intermediate nodes and transferred over different configurations of the MultiRing. Message routing problems can be solved in *O (log N)* or *r* configurations of the network. This means any two nodes can communicate within one complete cycle of all configurations of the switch.

A restrictive case may exist where a node can send data only to its *right* neighbor on a ring. At a given configuration, the node can choose to either send a message to its right neighbor or keep the message to send at a later configuration. After *r* configurations, a node is able to contact all other nodes. Figure 1.5 illustrates how P_0 sends a message in an 8-node MultiRing to all other nodes in three configurations. The tree diagram represents the routing options available at a given configuration. A node can either send a message to its right neighbor (straight arrow) or keep the message (dashed arrow). At the end of the 3rd configuration, node P_0 has contacted all nodes. The dashed lines represent the case where the message is not sent during a configuration. (In this example, the configuration with 8 rings of 1 node is not needed.)

A complete cycle of the switch occurs in this example, when the network is configured first into 1 ring of 8 nodes, then into 2 rings of 4 nodes and finally into 4 rings of 2 nodes. If P_0

wants to send a message to P_1, P_3, P_5, or P_7 it sends the message to P_1 while the ring is configured for 1 ring of 8 nodes; otherwise, it waits until another configuration. When the MultiRing is configured into 2 rings of 4 nodes, P_0 sends messages destined for P_2 or P_6 to P_2. Finally when the MultiRing is configured into 4 rings of 2 nodes, P_0 can send P_4's message directly to P_4.

If a node receives a message that has not yet reached its destination, it forwards the message to its right neighbor on one of the remaining configurations in the cycle of the switch. For example, P_1 forwards messages destined for P_3 or P_7 to P_3 when the configuration is 2 rings of 4 nodes and forwards messages to destined for P_5 on 4 rings of 2 nodes. P_3 forwards messages for P_7, and P_2 forwards messages for P_6 on 4 rings of 2 nodes. Eventually, any message initiated by P_0 will reach its destination by the last configuration of the switch.

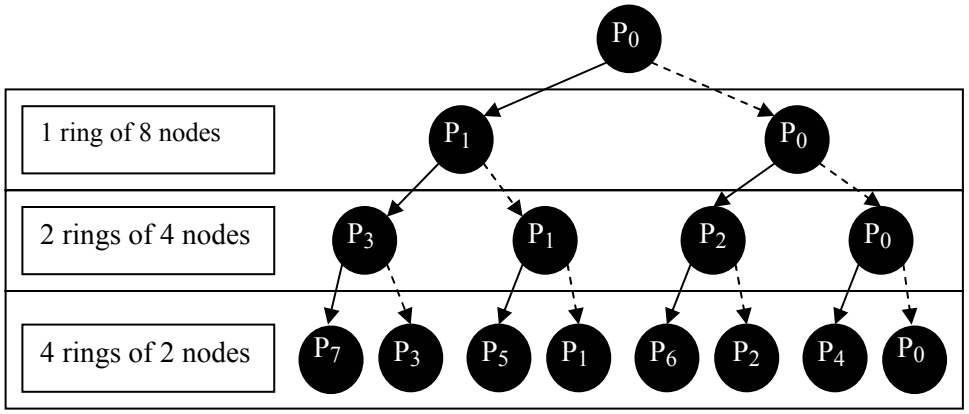


Figure 1.5: Routing messages through intermediate nodes on a MultiRing

In comparison with existing static ring networks, a MultiRing routes messages through fewer intermediate nodes because of the alternate paths produced at the different configurations. If only one static ring were used to connect the 8 processors forming 1 ring of 8 nodes, and each

node were restricted to sending data to only its *right* neighbor, when P_0 sent a message to P_7, these messages would have to be transferred through 6 other nodes before it reached its destination (P_1, P_2, P_3, P_4, P_5 and P_6). In contrast, on the MultiRing a message from P_0 to P_7 is transferred through 2 nodes (P_1 and P_3).

The time saved by having fewer intermediate nodes between communicating nodes is slightly offset by the time required to recognize the current configuration and to wait for the required configuration to transmit a message. Nevertheless, the MultiRing switch is designed to reconfigure the network quickly, and thus the reconfiguration time would be considered as negligible.

The design of the MultiRing Switch is similar to that of existing dynamic multi-stage interconnection networks such as the Butterfly and Omega networks; the switch is composed of a network of simple switching elements. To create the connections to form all rings available in an *N*-node MultiRing, *(N/2 * log N)* different switching elements are needed. However, in contrast to the Butterfly and Omega networks, the switching elements in the MultiRing are globally controlled to maintain multiple independent ring formations. Details of existing multi-stage interconnection networks are provided in Chapter 2 and a discussion of problems arising when attempting to map Butterfly and Omega networks to form the MultiRing are given in Chapter 3. The organization of the switching elements proposed by Arabnia and Smith [12] and a new design for the MultiRing switch are provided in Chapter 4.

1.3 Previous Research on the MultiRing

Arabnia and others have been researching the MultiRing since 1990 [4-18]. Over the years, these researchers have described several different algorithms for image processing and computer vision for use on the MultiRing [13-18] They have described and analyzed the

MultiRing's mathematical properties and primitive parallel operations for communication, and a sample design of the VLSI layout of a part of the MultiRing switch has been published[12]. It has been proven that 2D grids and *n*-cube algorithms can be mapped onto the MultiRing, and it has been shown that MultiRings can support function parallelism, data parallelism, and control parallelism in SIMD and SPMD modes[15].

1.4 Contributions of This Research

The approach taken in this dissertation to designing and developing algorithms for the MultiRing is based on the concepts of a) allowing multiple programs to work together on the same network independently and b) being able to add processors to the network with minimal effort. This approach had not been taken before and has led to new switch design and new routing methods.

The key element in the switch design is to build the MultiRing recursively: combine rings of two processors to create a ring of 4 processors; combine rings of 4 processors to create a ring of 8 processors, and so forth. The switch proposed in this dissertation is designed to enable processors to be added to the network without disconnecting any of the previous connections to the switch. This has obvious advantages related to installation time for new processors. In addition, the new switch maintains pre-existing rings in the expanded MultiRing, thereby enabling programs that were originally implemented on a ring of N processors to continue using these same processors in the new MultiRing of 2N processors.

The new routing methods for transmitting messages focus on routing messages between nodes working only on the same program. Previously, the MultiRing has been described for use with a SPMD or SIMD approach in which all nodes are working on the same program. The new

routing methods described here are for a MIMD approach that enables multiple programs to work independently.

There are three main methods for routing: arithmetic, source-based and table-driven. With arithmetic (algorithmic) routing, the output channel is decided when a message arrives, based on some calculation that takes into consideration its current position in the network, original source node and destination. With source-based routing, the source node creates a header that predetermines the path of the message, and, as a message is routed, the switch strips off one bit. With table-driven routing, each node maintains the next 'step' for a message. This table can be computed arithmetically during initialization of the network. This dissertation describes a new method for arithmetically computing the route of the message based on current configuration of the MultiRing, destination, and current position of the message. This method can be used to create routing tables during initialization of the network if space is available. The routing method is original.

Different communication models that can be used at the high-level to develop programs specifically for use on the MultiRing or at the low-level for routing messages have been developed. The pipeline, cube and tree models in particular can be used to map existing algorithms developed for static networks onto the MultiRing. Also, being able to use a particular communication model for special routing operations, namely ring broadcasting, group broadcasting and distributing, may have some advantages. Group broadcasting has never been discussed for the MultiRing, and the ring broadcasting and distributing algorithms designed are different from previous algorithms. These algorithms are based on allowing messages to be routed through nodes working on the same program. If the previous broadcasting and distributing algorithms were used, messages would be routed through nodes not on the same sub-MultiRing.

A by-product of this dissertation is a MultiRing simulator that a programmer can use to create and test parallel algorithms. All of the MultiRing routines are stored in a header file, mr.h, which must be included in C program. Similar to what must be done with PVM and MPI [1], the programmer must use specific functions to initialize the MultiRing and send and receive messages. Over 50 functions have been defined.

Three examples of sorting algorithms that use the communication models defined in this dissertation are provided; these can be implemented and tested on the simulator.

Chapter 2 provides an overview of interconnection networks. Included in this chapter are examples of static networks ranging from linear arrays to hypercubes and of dynamic networks ranging from a simple bus to a multi-stage interconnection networks including the Butterfly and Omega networks and reconfigurable meshes.

Keeping in mind that it is always best to use existing technology rather than reinventing the wheel, Chapter 3 explores building the MultiRing out of existing interconnection networks with perhaps only a few modifications. The MultiRing is mapped onto the Butterfly and Omega networks. Issues that arise from utilization of the existing networks are presented.

Chapter 4 provides both the original design of the MultiRing Switch proposed by Arabnia and Smith [12] and a new design for the MultiRing Switch. The original switch design is satisfactory in that it provides point-to-point networking without contention. However, to expand the network to include additional nodes requires disconnecting the existing nodes from the existing switch. In the new switch design, nodes are grouped in pairs and it is possible to build larger switches without disconnecting the existing nodes from the existing switch. This feature allows existing ring formations to remain intact when expanding the network. It also has the

added benefit that now a MultiRing does not have to have exactly 2^r nodes. When adding processors to an *N*-node MultiRing, fewer than *N* processors can be added to the network.

Chapter 4 also contains a description of how to create a bidirectional MultiRing and the controls necessary to provide clockwise and counter-clockwise ring communication. In addition, algorithmic issues regarding manual and automatic switches are provided. With a manual switch, the node issues a request for a configuration; with an automatic switch, the switch automatically cycles through all of the configurations and remains at a configuration for a set time.

In Chapter 5, communication strategies are presented for simulating a static network while supporting multiprogramming on a network of workstations. This chapter provides models for simulating algorithms developed for pipeline, n-cubes, 2D grids and trees on the MultiRing. Each model includes strategies for choosing the best configuration to simulate the network based on the size needed for the algorithm, and the method of reconfiguring the network to allow multiple independent sub-networks. Communication strategies are different depending on whether only unidirectional links are used or if bidirectional communication is possible on a ring. In an unidirectional data flow, all nodes can send data to the right (left) neighbor during a configuration. In a bidirectional MultiRing, all nodes can send data to left and or right neighbor during a configuration. Bidirectional links are needed for n-cube, grid and tree algorithms.

Primitive routing operations are presented in Chapter 6. The MultiRing has a limited set of direct links, and routing algorithms are used to direct the flow of messages enabling each message to reach its destination with the shortest possible communication path. Ring broadcasting, group broadcasting and distributing are developed based on the pipeline, grid and tree communication models first described in Chapter 5. Depending on the operation, one communication model may yield a simpler algorithm.

This dissertation describes an approach to designing and developing algorithms for the MultiRing based on multiprogramming where multiple programs can execute concurrently on the same MultiRing using disjoint subsets of the MultiRing's nodes. Chapter 7 demonstrates how multiprogramming is accomplished on the MultiRing.

Chapter 8 described the MultiRing simulator that was created to test the routing and communication strategies. This simulator is useful in that other programmers can use it to create and test parallel algorithms.

Sorting is a common operation found in most computer applications. In Chapter 9, three different parallel sorting algorithms developed for the MultiRing are presented. Bitonic, MultiQuicksort and bin-collecting are based on algorithms originally developed for the hypercube. These algorithms illustrate how ring broadcasting, group broadcasting and distributing can be implemented on the MultiRing as a program executes.

Chapter 10 contains the conclusion of this dissertation and presents possible future work.

CHAPTER 2 : INTERCONNECTION NETWORKS

The interconnection network, referred to by Jordan and Alaghband as "the heart of a parallel processor" [37], is important because it impacts performance and scalability of a parallel computer and complexity of parallel programs. Various topologies have been designed to connect processors for inter-processor communication or connect processors and memory modules in shared memory systems, each being especially suited to certain situations.

Parallel computer performance speed is enhanced if communicating processors are connected directly to each other rather than indirectly connected. However, as programmers try to exploit communications paths of processors limited to direct connections, the complexity of programming increases. Furthermore a design that requires extensive rewiring when adding processors reduces the interconnection network scalability. If the number of links for each processor increases when a new processor is added, even by one, the network will not scale smoothly for increasingly larger number of processors, as a result.

Interconnection networks may have *static* or *dynamic* implementations. In a static network, processors are connected directly through links which do not change during a program's execution. In a dynamic network, processors are connected via a collection of switches that can be reconfigured to meet communication needs during the execution of a program.

The MultiRing is an interconnection network that provides efficient processor communication with a limited number of links. This chapter provides a survey of a variety of interconnection network topologies and includes examples of existing parallel systems. Section

2.1 of this chapter provides common parallel processing definitions. Examples of static networks are in section 2.2 and dynamic networks are in section 2.3.

2.1 Definitions

This section contains definitions of terms in parallel architecture that are used throughout this paper.

Parallel computer architecture classifications include *SIMD*, *MIMD* and *MISD,* the architectures being distinguished according to the level of data and program execution control independence. SIMD (single instruction, multiple data) computers have a single control unit with a number of processing elements (PEs). The control unit issues the same instruction to all PEs, but each has different data. SIMD machines support lock-step processing in which the same operation is performed simultaneously in each PE.

MIMD (multiple instruction, multiple data) computers are also called *multiprocessors*. A multiprocessor is a single system that contains multiple processors that work independently but are networked for inter-processor communication or to interface with memory and I/O devices. On MIMD architectures, each processor can execute a different program.

MISD (multiple instruction, single data) computers execute different programs on the *same* data item. This implies that several instructions are operating on a single piece of data. Pipelined architectures such as vector processors and systolic arrays are considered MISD. Data flows through a series of stages, each stage performing a particular function and producing an intermediate result. On MISD computers, elements of a vector belong to the same piece of data and all pipeline stages represent multiple instructions that are applied to the vector.

Regardless of the computer architecture's classification, some form of interconnection network is needed by all parallel architectures. The interconnection network allows the

processors to communicate either with each other or with memory or I/O devices. In shared memory systems, the interconnection network is between processors and memory so that a processor can access any memory location. In distributed memory systems, each processor has its own memory and the interconnection network is used to transmit data directly between processors.

2.2 Static Networks

Common static network topologies include linear array, grid, ring, hypercube and fully connected networks (Figure 2.1). "The fixed communication topology of static networks makes them most appropriate for problems with predictable communication patterns involving mostly neighboring processors" [37].

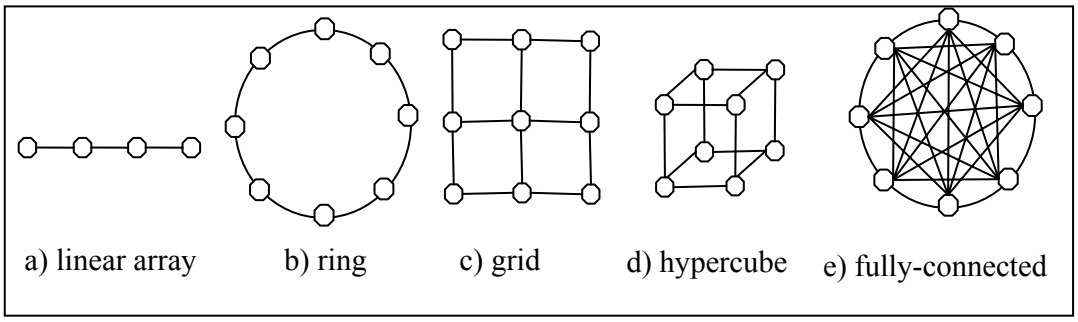


Figure 2.1: Common static network topologies.

In general, one network topology is more desirable than another if it is more efficient, more convenient, and more extendable than the others. We use three basic metrics related to communication complexity to evaluate static networks: 1) diameter of the network, 2) connectivity and 3) bisection width.

Diameter is defined as the shortest path connecting the two processors which are the farthest apart in a network. Establishing a small diameter provides a lower bound for the communication complexity of algorithms implemented on a network that requires information sharing among the processors. As a general rule, a network with a smaller diameter has a larger number of links to each processor in the interconnection network.

Connectivity is measured in terms of the maximum node degree in the network. Node degree is defined as the number of links or communication ports to the processors. The maximum degree of a network is determined by the degree of the node in the network with the most links.

Bisection width is equal to the smallest number of links so that when they are removed, the network is split into two pieces each having the same number of nodes. If the bisection of a network with *2N* processors is *B*, two networks with *N* processors each can be combined into a *2N* network by adding *B* links to connect the two networks.

Scalability is the ability of a network to expand to include a large number of processors and increase the performance proportionally. In general a network with low diameter, connectivity and bisection width is considered scaleable. In the following sections a few common static networks are examined and their diameter, connectivity and bisection width are compared.

2.2.1 Linear Arrays

The simplest form of interconnection network is a linear array (Figure 2.2). In this topology, *N* processors are connected from point to point in a row using *N-1* bidirectional links. Linear arrays are characterized by having a high diameter, low connectivity and low bisection width. The diameter of a linear array is *N-1* – the distance between the first and last processors in

the array. This is rather high for large *N*. The maximum degree of the network is 2 since excluding the first and last processors, each processor connects to two neighboring processors. The removal of a single link partitions the network; therefore, the bisection width is 1.

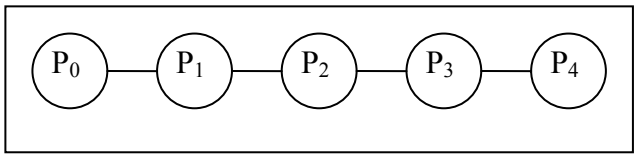


Figure 2.2: Linear array of *N=5* processors.

In linear arrays, there is exactly one route for communication between pairs of processors. Long communication delays may result as data is transmitted through a number of intermediate processors before reaching the final destination. Consequently, communication can become inefficient when *N* becomes large, especially between the first and last processors. However, it is possible on a linear array for different pairs of processors to communicate *simultaneously*. Processors can use *disjoint* sections of the network concurrently. For example, in Figure 2.3, P_0 communicates with P_2 indirectly through P_1 while at the same time P_4 communicates with P_3.

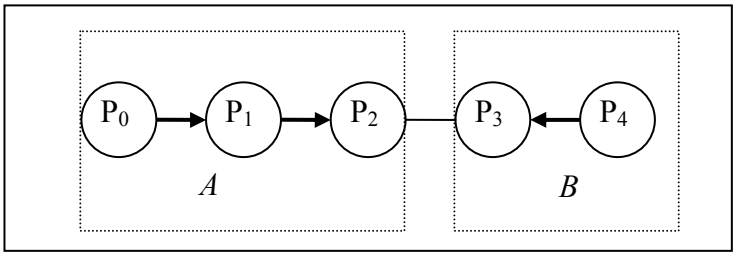


Figure 2.3: Disjoint sections of a linear array are used concurrently.

2.2.2 Ring

Rings are closely related to linear arrays. Essentially, the first processor in a linear array is connected to the last processor (Figure 2.4). Adding that extra link reduces the diameter of the network to *N/2*. Even though the diameter is half that of a linear array, it is still considered high. The connectivity and bisection width remains low with a maximum degree and a bisection width of 2.

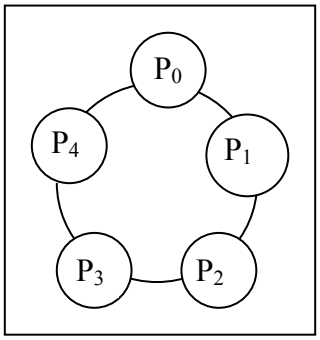


Figure 2.4: Ring of ***N***=5 processors.

The simple routing and low hardware complexity of rings have made them popular for local area networks. For example, many computer labs are networked on the IBM *Token Ring* which circulates a message around the ring of processors until it reaches its destination. Only one processor can transmit a message at a time. This processor holds the writing "token." Each processor receives the token in turn. The ring inherently supports broadcast-based communication since whenever a processor receives a message, it inspects (snoops) the transmission before forwarding it, if necessary, to the next processor in the ring. As a result, one message can be "broadcast" to all processors [26].

An example of a parallel computer with a ring interconnection network is the Kendall Square Research KSR-1 machine developed in 1992 [39,40]. The KSR-1 is an MIMD computer

with 1088 processors each with 32 MB of local memory. It is a virtual shared memory architecture where memory is physically distributed among the processors but logically shared. If a processor cannot find data in its local memory, a request for data is sent to other processors via the interconnection network. Processors are logically grouped into different levels and requests for data are sent to processors at higher levels only if they are not found at a lower level.

A variation of a ring of $N=2^r$ processors is the barrel shifter. Each processor connects to other processors that are 2^i links away for all i between 0 and $r-1$. A barrel shifter has a node degree of $2r-1$ and a diameter of $r/2$ [35,37].

2.2.3 Grid

A grid (mesh) is a popular high dimensional array topology implemented on various parallel architectures including Goodyear MPP and Intel Paragon [35]. Compared to linear arrays and rings, grids have a low interconnection diameter but higher connectivity and bisection width. In general a k-dimensional grid with n nodes on each dimension has an interior degree of $2k$ and a diameter of $k(n-1)$. For example, in Figure 2.5, a two-dimensional grid with three nodes on each dimension has an interior degree of 4. The diameter is 4, since the shortest route from P_0 to P_8 requires transmitting data over 4 links. A benefit from the increased connectivity is that alternate communication paths between two processors are available.

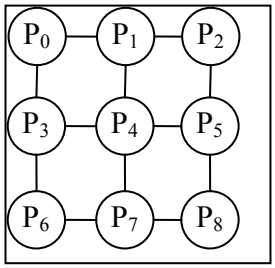


Figure 2.5: 3x3 two-dimensional grid.

A torus extends the grid by having wraparound connections that reduce the diameter of the grid by half. The torus is a symmetric topology. A two-dimensional grid with torus connections was one of the first examples of parallel architecture implemented – the Illiac IV. The Illiac IV was built in 1972 for the SIMD solution of partial equations. The original designed called for 4 quadrants of 64 processors; however, only one 8X8 quadrant was created in the prototype [32,35]. As illustrated in Figure 2.6, the Illiac IV has a node degree of 4, a diameter of 7 and bisection width of 8.

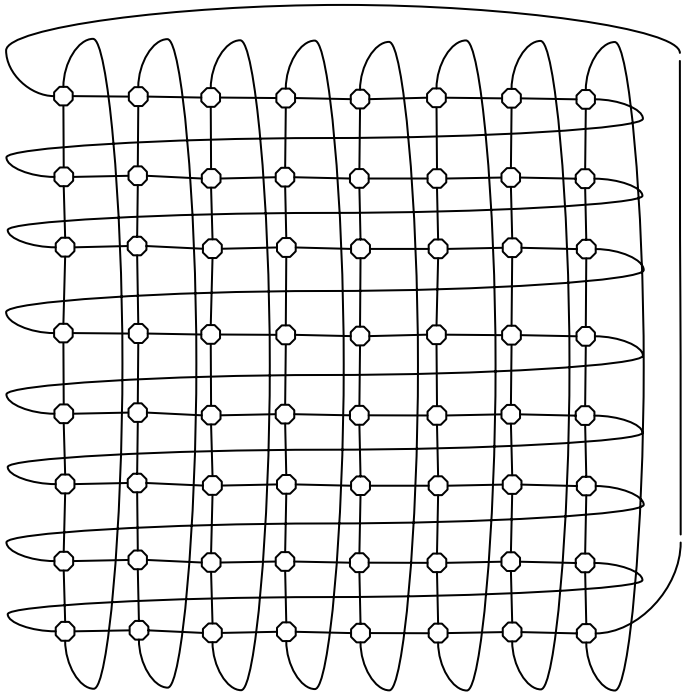


Figure 2.6: Illiac IV -- an 8X8 grid with modified torus connections.

Pipelining, one way of programming so processors operate in parallel, is often implemented on linear arrays and grids. Data flows through a series of stages; at each stage a particular function is performed and an intermediate result is produced. At the last stage in the

pipeline the final result is produced. Parallelism is achieved when new operations are initiated at the start of the pipeline while other operations are in process through the pipeline. Even though pipelining can be implemented on other computer architectures, MISD architectures, namely vector processors and systolic arrays, have been designed with pipelining as the principle concept.

A vector is an ordered set of elements, and many applications exist where the same function is applied to each element in a vector. Vector operations permit more parallelism to be obtained with a single thread of control; a single instruction can initiate a long vector operation. A vector processor with a pipelined structure has a pipeline of PEs with each PE performing a single stage of vector arithmetic. A floating point add, for example, is not done in one clock cycle; it may take 4 steps. Therefore, a collection of PEs may link to form a pipeline adder (Figure 2.7).

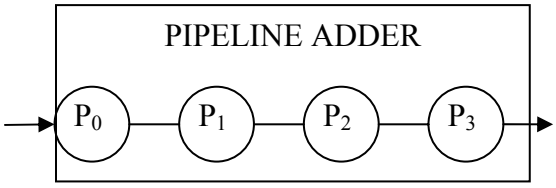


Figure 2.7: Pipeline adder with four stages.

If A, B and C are vectors and the vector instruction $A+B=C$ is issued, each pair of elements in A and B would be added to produce a value for the corresponding element in C. In Table 2.1, elements 0 and 1 of the vectors A and B traverse through the pipeline. The pipeline adder receives as input the element pairs (A_0, B_0) and (A_1, B_1) sequentially, vector operations overlap, and C_o and C_1 are produced, respectively, in 4 and 5 cycles.

Table 2.1 Overlapping vector operations in the pipeline adder.

Cycles	Stages of Pipeline Adder				
	P_0	P_1	P_2	P_3	
0	(A_0, B_0)				
1	(A_1, B_1)	(A_0, B_0)			
2		(A_1, B_1)	(A_0, B_0)		
3			(A_1, B_1)	(A_0, B_0)	
4				(A_1, B_1)	C_0
5					C_1

Two examples of vector processors are CDC STAR 100 and Cray I. The design of the CDC STAR, the first pipeline computer, was started in 1965, but it wasn't completed until 1974. The Cray I was the first commercially successful pipeline computer. In addition to pipelining, the Cray I had a hierarchical memory structure featuring eight vector registers which increased the speed of vector operations by reducing contact with main memory [56].

Array and vector processors are sometimes considered to be SIMD architectures since the computations can be performed in lockstep fashion [32,37]. However, Roosta [54] considers vector arrays to be MISD architectures where the elements of a vector belong to the same piece of data and all pipeline stages represent multiple instructions applied to the vector. An illustration of the difference between SIMD and MISD, would be adding vectors that have four elements where floating point addition is carried out in four cycles. A true SIMD with four PEs would compute four additions simultaneously in four cycles (Figure 2.8a). In SIMD architectures, each processor performs a *complete* floating point addition. In contrast, in MISD

architectures, each processor performs one stage of a floating point addition. A vector processor with a 4-stage addition pipeline would compute four additions in 7 cycles using 3 cycles to fill the pipeline plus 4 more cycles to deliver all of the results (Figure 2.8b).

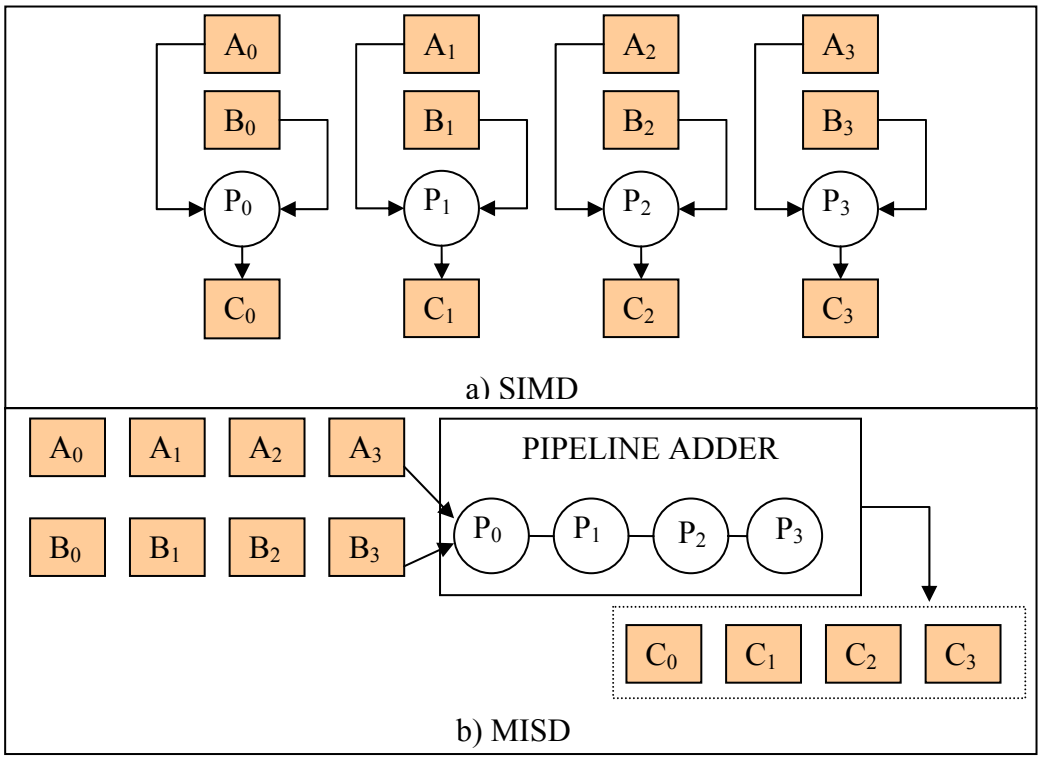


Figure 2.8: SIMD and MISD vector processors.

Systolic arrays are a special class of application driven array architecture pipelined with multi-directional flow of data streams. Systolic arrays differ from the conventional pipeline designs by incorporating triangular and hexagonal interconnections in addition to the standard grid interconnections (Figure 2.9). The pathways between PEs are multidirectional and, in contrast to SIMD, each PE can perform a different operation.

Systolic arrays have been used in special applications like image processing; however, they cannot implement other algorithms efficiently once having been optimized for a given

application. In general, they are of limited application and are difficult to program [35,68]. The Wrap, a one-dimensional systolic array built in the mid 1980s, and the iWrap, a two-dimensional systolic array built in 1992, were the first two systolic arrays [54].

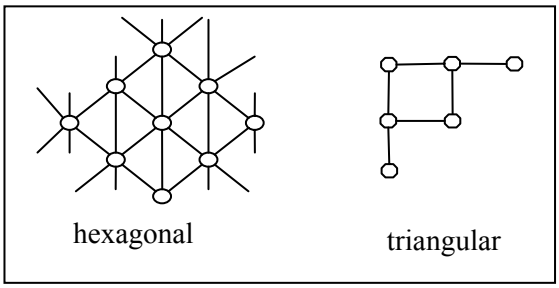


Figure 2.9: Hexagonal and triangular topologies of systolic arrays.

2.2.4 Hypercube

The concept of the hypercube is based on an *r*-bit node ID [41]. An *r*-dimension hypercube has $N=2^r$ nodes, each with a unique r-bit string as its node ID. For example, a three-dimensional hypercube has 8 nodes (i.e. 2^3 nodes) with the following node IDs: `000`, `001`, `010`, `011`, `100`, `101`, `110` and `111` (Figure 2.10). Each node is connected to neighbors, each of whose node ID formed by inverting a single bit of its own ID. For example, nodes `010` and `000` are linked because their *r*-bit strings differ by exactly one bit. Furthermore, node `010` is also connected to nodes `110` and `011`.

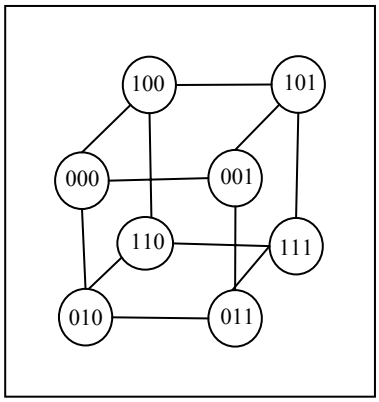


Figure 2.10: Three-dimensional hypercube.

The hypercube has a bisection width of *N/2*. The hypercube is bisected into two cubes by removing *r* links of an *r*-dimension hypercube to have two *r-1* dimension hypercubes. The node ID string is arranged in the sequence "123…*r*" where the first bit represents the 1st dimension links, the second bit represents the second dimension links and so on until the r^{th} bit represents the r^{th} dimension links. The removal of *r*-dimension links will leave two disjoint *N/2* hypercubes. In Figure 2.11, the removal of the 3rd dimension links from a three-dimensional hypercube forms two 2-dimensional hypercubes.

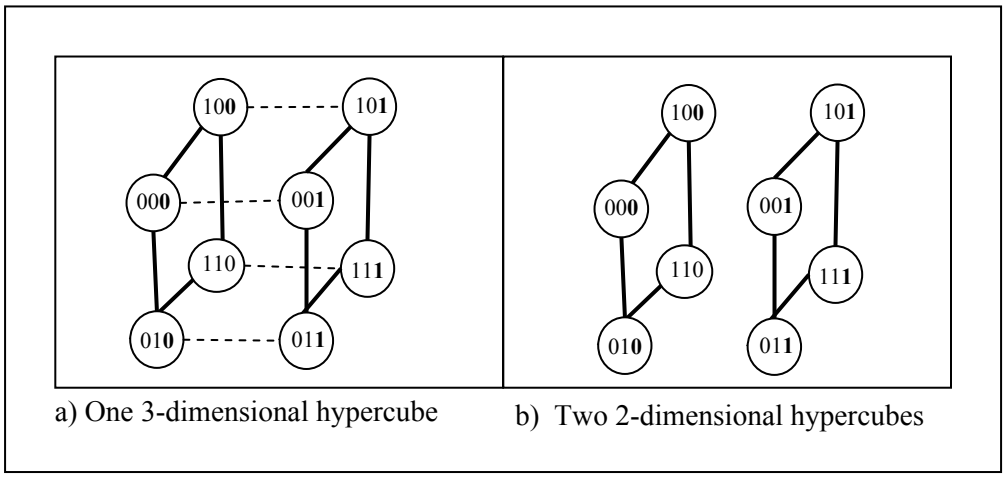


Figure 2.11: Bisecting a 3-dimensional hypercube into two 2-dimensional hypercubes.

In a hypercube, as in a grid, there are different ways to route message messages. For example one possible path to connect 000 to 111 is: 000-100-110-111. Another possible path connecting 000 and 111 is 000-001-011-111. In contrast to a grid, the diameter of a hypercube is *log N*. Messages can be routed between any two nodes in *log N* steps. "The diameter is smaller than a grid if there are more than 4 processors" [2].

The hypercube is a powerful network because it can simulate a variety of networks. For example, a linear array of *N* nodes is formed by creating a path that connects all nodes without repeats. Data simply traverses the nodes in the sequence of a Hamiltonian cycle in which the gray code ordering of all node IDs differ in 1 bit position [41]. Parallel algorithms designed for other network topologies can be implemented on a hypercube without many changes. The hypercube may be used for both special and general purpose tasks.

Hypercubes, common in the 1980s, were the basis for many real machines. In Table 2.2, examples of both SIMD and MIMD hypercubes are provided. Today, they have been replaced with lower dimensional topologies because wiring complexity and packaging proved to be too costly [37].

With hypercubes, the degree of each node grows logarithmically as the number of the nodes in the network increases. As a result, processors designed specifically for an *N*-node hypercube cannot be used in a *2N*-node hypercube. A 10-dimensional hypercube has 1024 nodes each with 10 neighbors; whereas a 20-dimensional hypercube has 1 million nodes each with 20 neighbors. An *r*-dimensional hypercube has $r2^{r-1}$ links.

Table 2.2: Examples of existing hypercubes.[1]

Machine	Year	Machine Type	Number of Nodes
CM-1	1983	SIMD	4096
CM-2	1985	SIMD	65536
Cosmic Cube	1983	MIMD	64
NCube	1985	MIMD	1024

2.2.5 Fully Connected Networks

In a fully connected network, each processor is connected directly to all other processors. The diameter is the lowest of all static networks, 1; however, the connectivity and bisection width are the highest of all static networks. The degree of each node is $N-1$ and the bisection width is $(N/2)^2$. Completely connected networks are only used with a small number of processors since direct connection between all pairs of processors is infeasible in a large network. O (N^2) wires are required for a fully connected network.

2.2.6 Summary of Static Networks

Static networks connect processors directly using fixed links. In this section, linear array, ring, grid, hypercube and fully connected networks have been described. The point is that an interconnection network should connect processors for efficient processor communication; yet at the same time to be useful in a laboratory, the network is expected to be scaleable. It is clear that a static network fits within a spectrum of networks ranging from interconnections that provide direct communication between each pair of processors to networks that are scaleable, because only one physical link is needed to add a new processor.

[1] SOURCE : Seyed H. Roosta. *Parallel Processing and Parallel Algorithms: Theory and Computation*. Springer, pages 80-81 2000.

On the other hand, a fully connected network allows all processors to connect directly with each other; yet this network can only be used with a few processors since it does not scale well. Adding even one processor requires extensive rewiring – connecting the new processor to the existing network requires adding a physical link from all existing processors to the new processor.

Linear arrays are seen to represent the other end of the spectrum with a limited number of links. Only one additional link is required when a new processor is added to the network. However, communication is not efficient since messages may have to be routed through intermediate processors.

The hypercube was created as a compromise between a fully connected network and a linear array. On a hypercube, the communication path and number of links are logarithmic; yet adding even one link from each existing processor to a new processor when increasing dimensions proves too costly.

Dynamic networks address the scalability issues found in static networks. In the next section, dynamic networks are explored as ways to indirectly connect processors while still maintaining efficient inter-processor communication.

2.3 Dynamic Networks

Static networks yield the best performance when the communications between processors mimic the network topology. However, for some applications, communication cannot be limited to only neighboring processors. Applications that involve different patterns of communication between processors as the program executes are best implemented on a dynamic network. A dynamic network can be reconfigured to handle different communication patterns based on a

program demands. Jordan and Alaghband describe dynamic networks as "indirect networks that provide great flexibility in communication patterns" [37].

Processors are connected indirectly through one or more switches. In contrast to static networks, like grids and hypercubes, a processor has a fixed number of communication ports which will not increase as the network doubles in size. In dynamic networks, the number of switches, not the number of communication ports, fluctuates depending on the number of processors in the network. In a dynamic network the degree of each processor is fixed at *N*. Processors are connected to *all* other processors.

Dynamic networks vary based on switching and wiring complexity. A few variations include:

- Number of switches
- Number of inputs and outputs for switch
- Number of control signals required to manage network
- Control strategy (distributed or synchronized)
- Interconnection pattern for connecting switches

Bus systems, crossbar switch networks and multi-stage interconnection networks are examples of dynamic networks discussed in the following sections. The metrics used to compare these networks include diameter, number of switches, concurrency and rewiring cost. The diameter is the width of the switching network. A low diameter is best since a *wide* network results in higher communication latency. *Concurrency* is the number of independent connections that a network can make between a pair of processors at a given time. A high level of concurrency is wanted in a parallel system. The rewiring cost is based on the number of links

that have to be rewired or added to double the size of an existing network. A low rewiring cost leads to modular scaling of the network.

2.3.1 Bus

A bus is the simplest form of dynamic network. A bus uses a single means of communication which is time-shared among all processors. Switches are used to establish links between two processors connected by the bus. At first glance, a bus may seem like a linear array, but in a bus all processors can communicate directly with each other; whereas with a linear array, multiple steps are required to route messages between non-adjacent processors. As Figure 2.12 illustrates, on a linear array processors can simultaneously write to neighboring processors using disjoint sections of the network concurrently. However, writing access is controlled by switches to ensure that only one processor can write to the bus at a given time. As a result, concurrency on a bus is 1. Allowing only one message to be routed at a time can lead to a major problem -- contention.

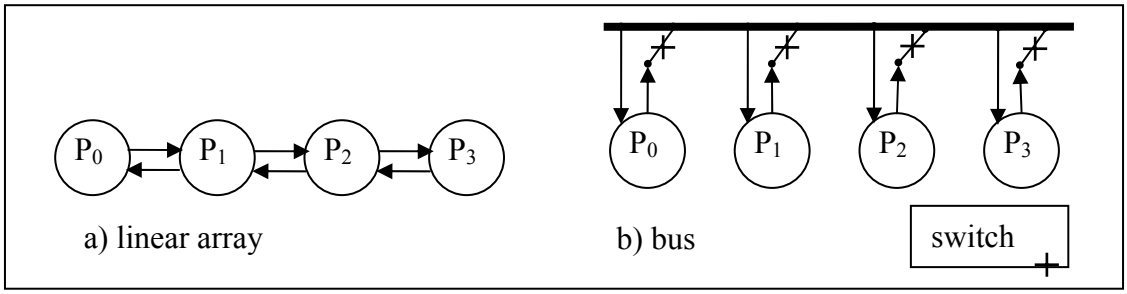


Figure 2.12: Linear array versus Bus.

The diameter and bisection width of a bus is 1 and the degree is *N.* An advantage of using a bus is that the loss of one link removes only one processor from the network [26]. Also, even though only one message can be routed at a time, each processor can concurrently observe every

transaction. Therefore, broadcast-based communication latency is lower than on a linear array or ring.

The rewiring cost for doubling the size of the network is low , requiring the addition of N more links, one for each new processor. However, the clock rate of a bus decreases with the number of additional processors since the time for two processors to communicate increases with distance. Therefore, busses are commonly used for a small number of processors [26,37].

The Ethernet used in local area networks is essentially a bus, only longer. The network is a basically a single, shared wire. However, unlike a bus, the Ethernet has no explicit switch control to guarantee one transmission at a time. A processor checks to see if the network is quiet and optimistically sends the message. As a result, there are collisions of messages on the Ethernet that are not possible on a bus.

2.3.2 Crossbar

The crossbar is the dynamic implementation of a fully connected network. Each processor can communicate directly with any other processor in the network. In fact, in a network of N processors, concurrency is N , that is, N messages can be transmitted in parallel. The only constraint is that no two processors P_i and P_j can send messages to the same processor P_r in parallel (i.e. P_1 and P_2 cannot both transmit to P_0 at the same time).

Each processor has N switches which control access to connecting processors for distributed memory architectures or memory modules for shared memory architectures. Thus, the crossbar is an expensive network, requiring N^2 switches. The diameter of the crossbar is 1; there is only one switch in a communication path connecting a sender and receiver. A 4X4 crossbar network is shown in Figure 2.13. Only one switch in a column is active to guarantee a receiving processor can receive data from only one sending processor at a time.

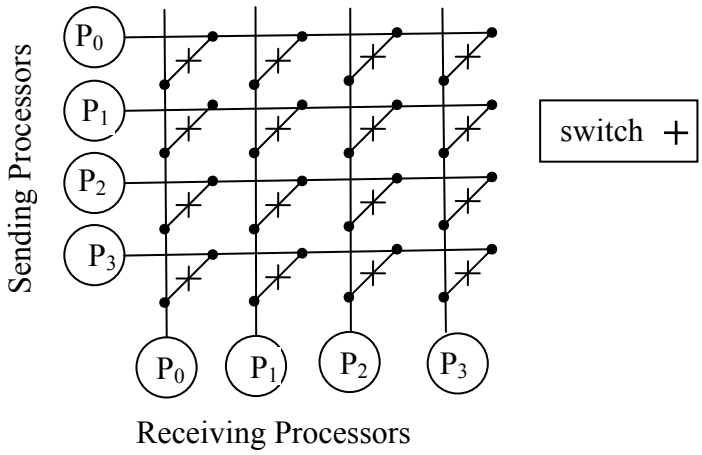


Figure 2.13: 4X4 crossbar network.

The cost of rewiring a *N*x*N* crossbar into a *2N*x*2N* crossbar is high since $3N^2$ additional switches are required. Although *NxN* crossbars are not practical when *N* is large, a group of smaller *rxr* crossbars where $r < N$, are often used to build larger interconnection networks. For example, IBM developed the Scaleable Power parallel 1 (SP1), a cluster of 64 workstations. The SP1 uses a multistage interconnection network based on an 8x8 crossbar switching element [28,35]. In fact, all multi-stage interconnection networks which are described in the next section use a collection of 2x2 or larger crossbar switches.

2.3.3 Multi-Stage Interconnection Networks

In a multi-stage interconnection network (MIN), processors are connected indirectly through a series of crossbar switching networks. In Figure 2.14, different source and destination IDs are given; but, due to wrap around, when connecting processors to processors, processor S_0 is also D_0 on some MINs. In shared memory systems, the sources may be processors and the destinations could be memory modules.

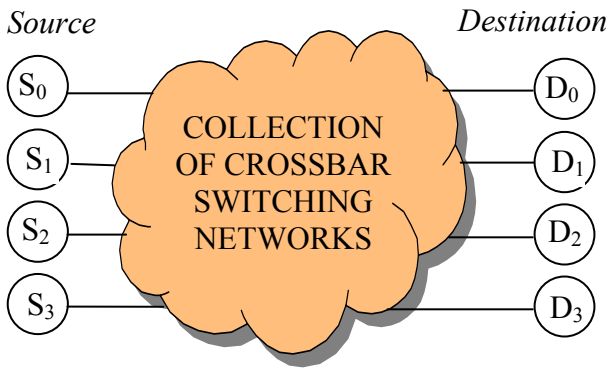


Figure 2.14: Multi-Stage Interconnection Network.

Provided each processor can send and receive a message simultaneously, concurrency in a multi-stage interconnection network is *N*. The goal of a multi-stage interconnection network is to allow *N* messages to be sent in parallel without requiring N^2 switches as required in an *NxN* crossbar switching network. A MIN generates different permutations (arrangements) of source/destination pairs where no two source processors send a message to the same node. Jordan and Alaghband state "A permutation represents coordinated transfer of data from multiple sources to the same number of destinations in parallel" [37].

All MINs use 2x2 or larger crossbar switching networks. A 2x2 crossbar switching network, also called an exchange element, has four states: bar, cross, upper broadcast and lower broadcast. Each exchange element contains four switches. In Figure 2.15, the different stages of the 2x2 crossbar and the associated exchange element are depicted. Each exchange element in a multi-stage interconnection network can be set independently and dynamically to establish a desired connection of source/destination pairs.

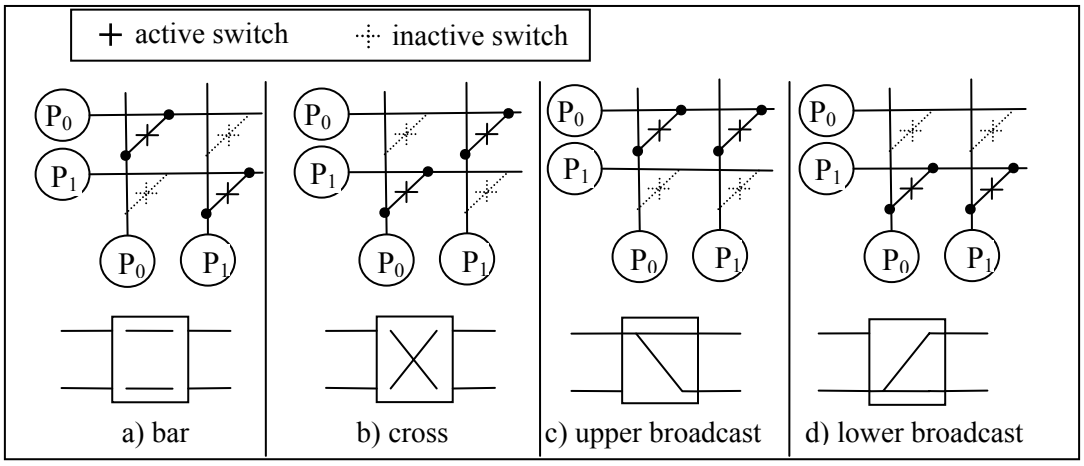


Figure 2.15: States of 2x2 crossbar and the corresponding exchange element.

The collection of crossbar switches in an MIN network is arranged in rows and columns and fixed inter-stage links connect the output of one column to the input of the adjacent column. MINs differ in the inter-stage connections. Two commonly used inter-stage connection patterns are the perfect shuffle and the butterfly.

The perfect shuffle connects processors according to a permutation that corresponds to shuffling a deck of cards [2, 37, 56]. Picture a deck of cards numbered from 0 to 7. Cut the deck evenly into two stacks with one stack having cards 0-3 and the other having cards 4-7. If shuffled perfectly, the new order will be 0-4-1-5-2-6-3-7. A perfect shuffle *f(x)* is defined mathematically as a permutation of $N=2^k$ integers where:

$$f(i) = \begin{cases} 2i & \text{for } i < N/2 \\ 2i-N+1 & \text{for } i \geq N/2 \end{cases}$$

The inverse perfect shuffle can be obtained by reversing the arrows of the perfect shuffle. Formally, the inverse perfect shuffle is defined as a permutation, $f^{-1}(x)$ of $N= 2^k$ integers where:

$$f^{-1}(i) = \begin{cases} i/2 & \text{for } i \text{ even between 0 and } N\text{-}2 \\ (i\text{-}1)/2 + N/2 & \text{for } i \text{ odd and between 1 and } N\text{-}1 \end{cases}$$

The perfect shuffle and inverse perfect shuffle of 8 integers are illustrated in Figure 2.16.

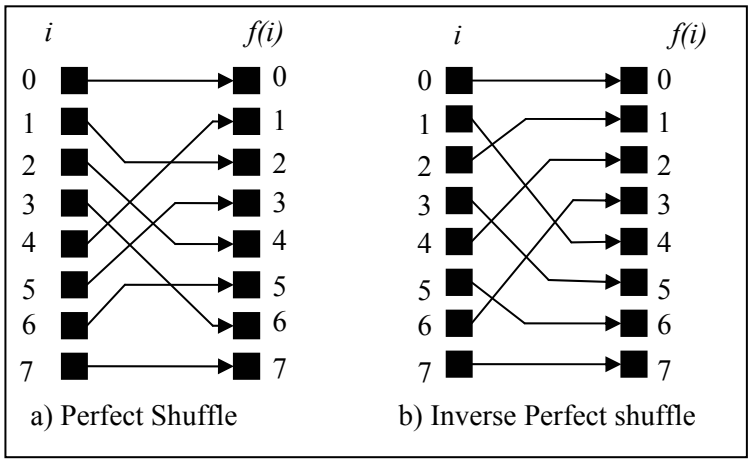


Figure 2.16: Perfect and Inverse Perfect Shuffle

The butterfly interconnection pattern connects a grid of nodes where each node corresponds to a (*w, i*) pair where *w* is the binary representation of the row number and *i* is the column number. Nodes (*w, i*) and *(w', i')* are linked by a *bar* edge if *i'=i+1* and *w=w'* or a *cross* edge if *i'=i+1* *w* and *w'* differ only by the i^{th} bit [37]. In Figure 2.17, nodes on a 4x3 grid are connected using the butterfly interconnection pattern.

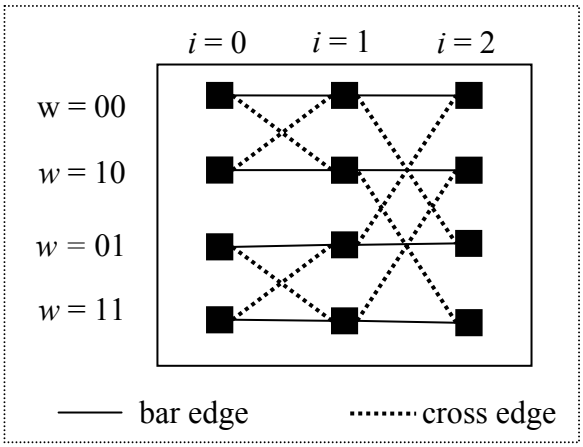


Figure 2.17: Butterfly interconnection.

The diameter of a MIN is *log N*; therefore, for transmission through the different columns (stages) of the switching network the latency of a MIN is *O(log N)*. In comparison, the diameter of a *NxN* crossbar is 1, and latency is constant. However, the cost of a MIN in terms of number of switches is *O(N log N)* compared to $O(N^2)$ for a crossbar. In addition, a MIN has modular construction which is beneficial when increasing the number of processors in the network. The rewiring cost when doubling the size of the network depends on the type of MIN used, but in general it is O(*N*).

Useful MINs can produce many different permutations by resetting the exchange elements. In a MIN that allows only bar and cross states, each exchange element can be controlled by a *single* control bit. If the control bit is 0, the exchange element is set to bar state. If the control bit is 1, the exchange element is set to cross state. There are *O(N log N)* exchange elements in a MIN, and changing the network to allow a different permutation may require changing the states of *all* switches in the network.

Determining the value of all of the control bits can be quite complex. Strategies for managing the controls are classified as either distributed or synchronized. Distributed control

requires “smart” exchange elements, each containing logic that determines its own control bit. For example, in section 2.3.3.3, a distributed method for communicating on the Omega network is described, where each exchange element receives the destination address as input and uses one bit of the address to set its state.

In synchronized control, each exchange element receives one bit from a global control word. The setting of all exchange elements is synchronized so that all exchange elements receive their control bit at the same time. Even though there are *O(N log N)* exchange elements, there doesn’t necessarily have to be a different control bit for each exchange element; some of the exchange elements share the same control bit. In most common MINs with $N=2^r$ processors, *r(r+1)/2* bits are used in the control word.

The Benes, Butterfly, and Omega MIN networks are commonly used because they can effectively simulate the static hypercube network. They differ in the number of exchange elements and the inter-stage connections (Figure 2.18); as a result they produce different permutations. The Benes produces *all* possible permutations where as the Butterfly and Omega networks only produce a subset of possible permutations. These networks are discussed in the sections 2.3.3.1- 2.3.3.3. All of the networks are illustrated with 2x2 exchange elements but other size crossbar switching networks can be used.

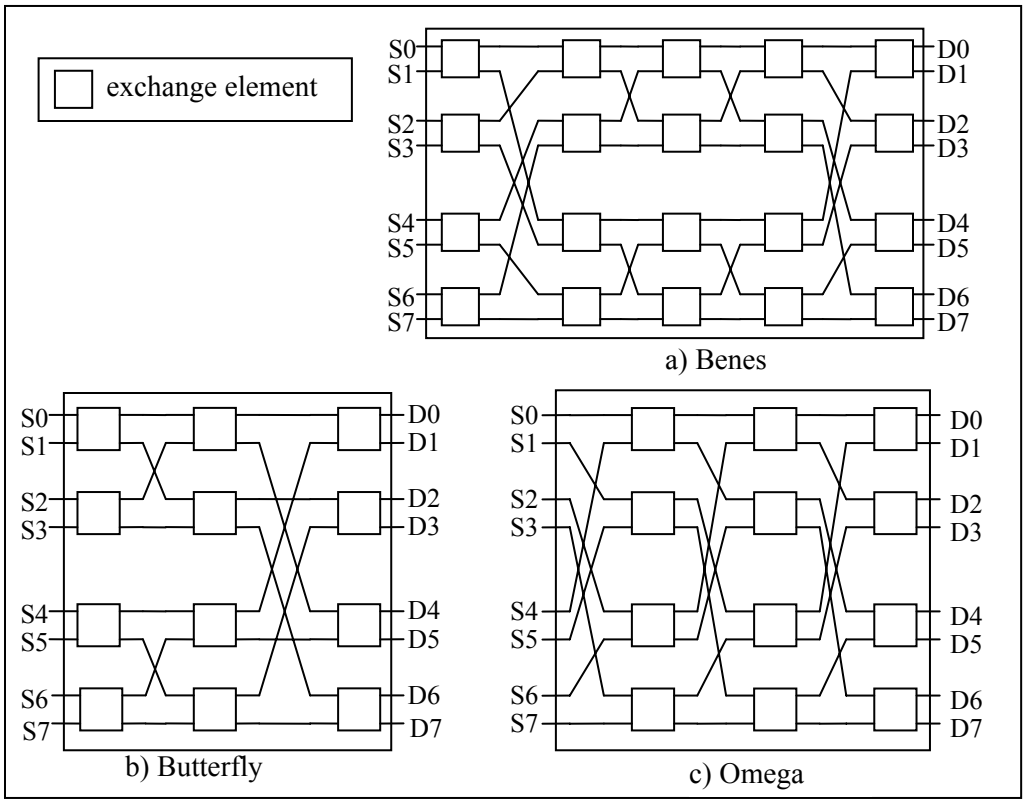


Figure 2.18: 8-Benes, Butterfly, and Omega multistage interconnection networks.

2.3.3.1 Benes Network

An *N*-Benes multistage interconnection network has *N* inputs and *N* outputs. It can produce all of the permutations available in a crossbar. The Benes network has *N/2* rows and $2 \log_2 N - 1$ columns of exchange elements. With 4 switches in each exchange element, there are $4N \log N$ switches, which, for large *N,* is less than the N^2 switches required by an *N*x*N* crossbar.

An *N*-Benes network can be recursively constructed with two *N/2*-Benes networks and *N* additional exchange elements (Figure 2.19). Both inverse perfect shuffle and perfect shuffle are used in the Benes network. Half of the new exchange elements connect to the inputs of the *N/2* networks using an inverse perfect shuffle. The remaining exchange elements connect to the output of the two *N/2* networks using a perfect shuffle. For example, in Figure 2.20, two 4-Benes

networks are used to construct an 8-Benes network. In general, the rewiring cost of constructing a *2N*-benes network with two *N*-Benes networks is *4N*.

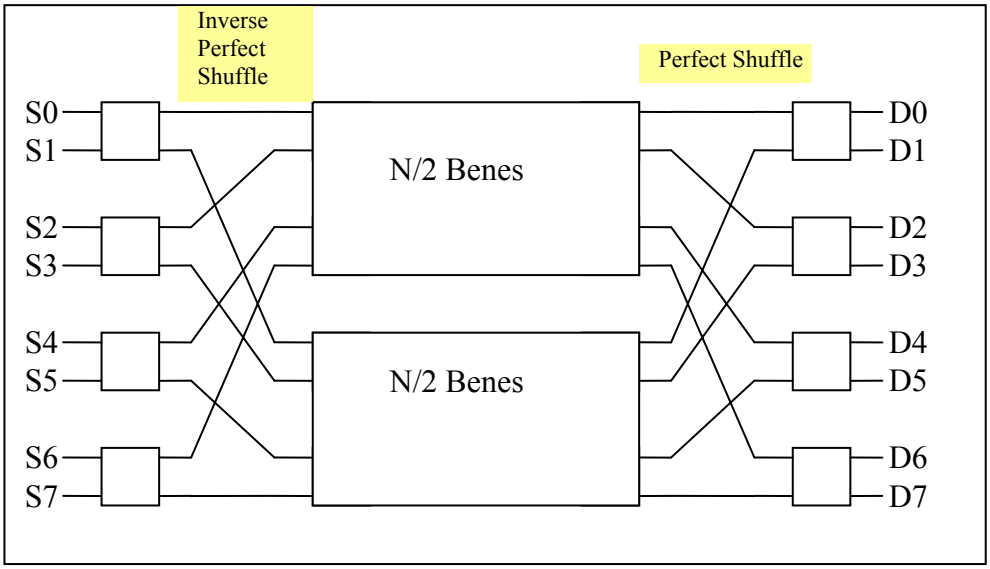


Figure 2.19: Recursive structure of the Benes network.

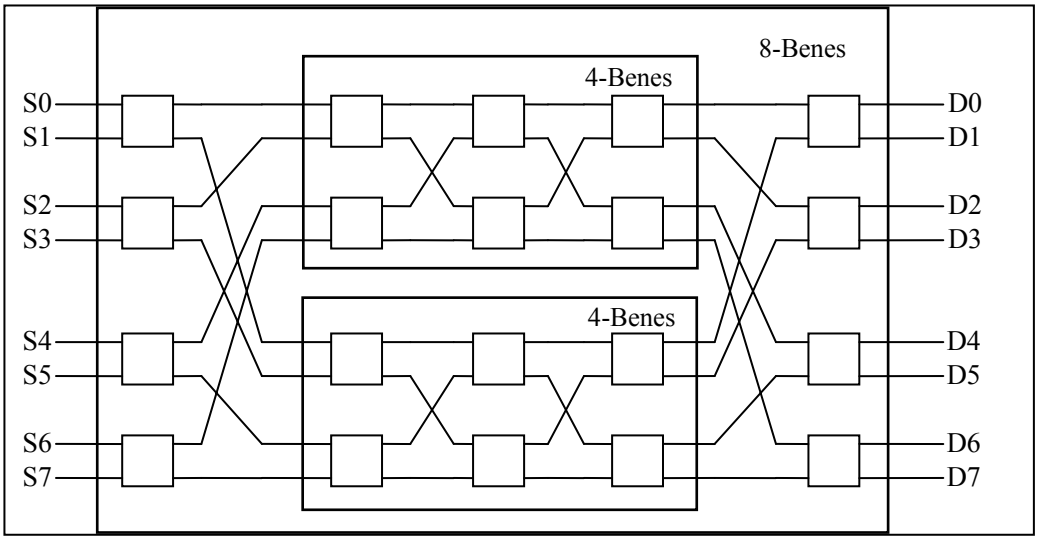


Figure 2.20: Constructing an 8-Benes network with two 4-Benes networks.

The N-Benes network is harder to control than an *NxN* crossbar switch. With crossbars, each pair of processors connects with *one* unique switch. Reconnecting a pair of processors involves disabling one switch and enabling another. In contrast, on a Benes network, the

reconnection of a pair of processors may require a complete reconfiguration of all of the exchange elements in the entire network.

The Benes network is useful but not all of the permutations it produces are required for some programs. The Butterfly and the Omega networks require fewer exchange elements and produce only a subset of the permutations; however, these permutations are ones that are commonly required by parallel programs.

2.3.3.2 Butterfly Network

The Butterfly network uses almost half the number of exchange elements of the Benes network. A *N*-Butterfly having $N=2^r$ processors is organized in *N/2* rows and *r* columns of exchange elements; this is $(N \log N)/2$ exchange elements for the Butterfly compared to $(N(2\log N - 1))/2$ for a Benes. The Butterfly MIN uses the butterfly interconnection pattern to connect adjacent elements, hence the name "Butterfly" network. An 8-Butterfly is shown in Figure 2.21.

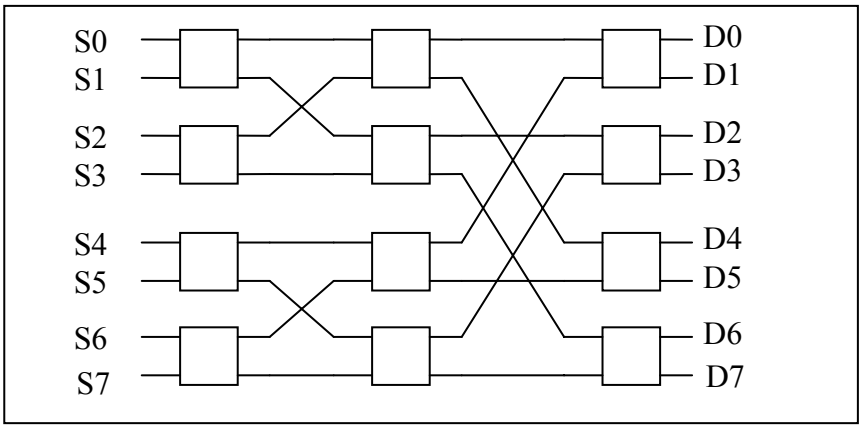


Figure 2.21: 8-Butterfly network.

Like the Benes, the Butterfly network is scaleable. In Figure 2.22, a *2N*-Butterfly is constructed with two *N*-Butterfly networks and *N* additional exchange elements. The outputs of the *N*-Butterflies are connected with the exchange elements according to the butterfly interconnection pattern using bar and cross edges.

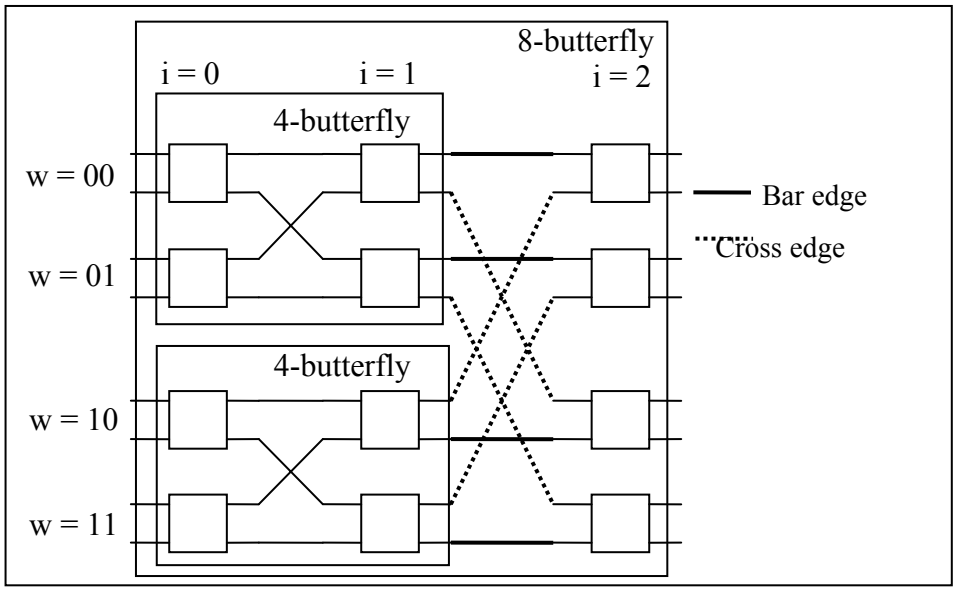


Figure 2.22: Constructing an 8-Butterfly network with two 4-Butterfly networks.

The GP1000, a commercial parallel processor manufactured by BBN Advance Computers Inc., uses the Butterfly multistage interconnection network [28]. The GP1000 multiprocessor is a shared memory system with 128 PEs connected by a high-speed butterfly switching network composed of 4x4 exchange elements. All memory in the system resides on individual nodes, but any processor can address any memory through the GP1000 Butterfly switch [45].

The dynamic Butterfly interconnection network can be implemented as a static network with fixed links by replacing the exchange elements with switching processors. In fact, any MIN

can be a static network when the exchange elements are replaced with processors dedicated to routing messages.

Figure 2.23 illustrates that a dynamic network can be converted into a static network by replacing switching elements with processors. As a result, the total number of processors in the network increases, with a possible increase in cost since a processor may be more expensive than an exchange element. In a static Butterfly the interconnection of $(r+1)2^r$ processors is divided into $r+1$ columns each containing 2^r processors. (Note: Roosta [54] and Leighton [41] describe the static network with the dimension of rows and columns interchanged. It consists of *r+1 rows* each containing 2^r processors).

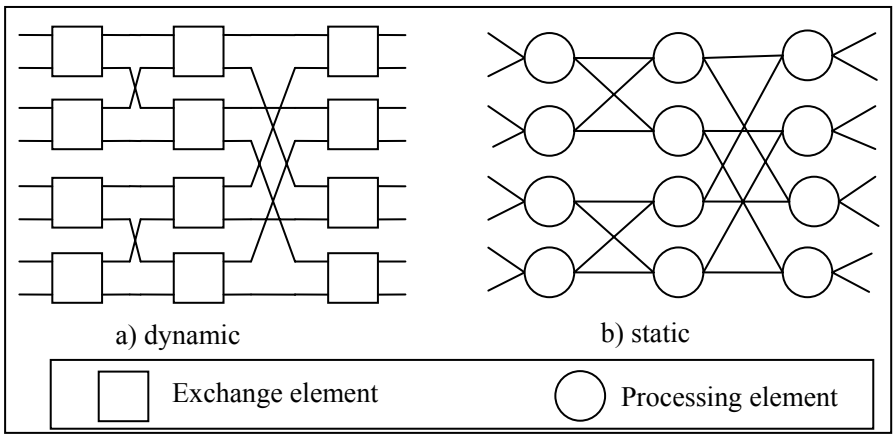


Figure 2.23: Static and dynamic Butterfly networks.

2.3.3.3 Omega Network

An *N*-Omega network connecting $N=2^r$ processors has the same number of 2x2 exchange elements as an *N*-Butterfly network, $((N \log N)/2$. However, the interconnection between stages is always a perfect shuffle (Figure 2.24). A single stage of an Omega network is a perfect

shuffle followed by an exchange element; leading to the designation of Omega networks as shuffle exchange networks.

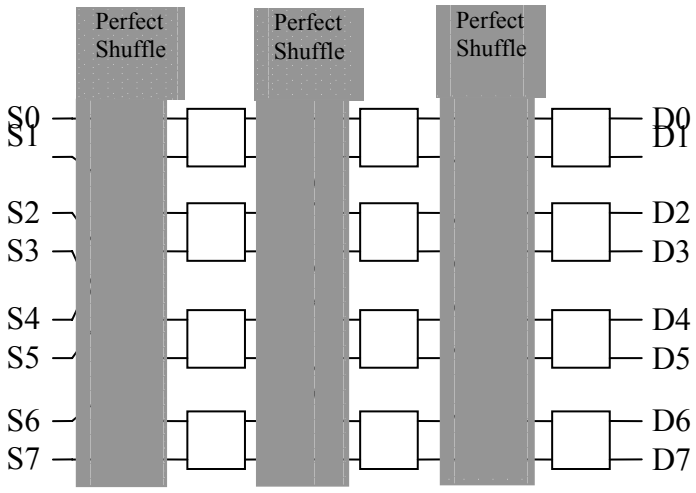


Figure 2.24: 8-Omega network

An Omega network with a distributed control scheme can route messages easily, using a simple routing algorithm based on the binary destination address. Each exchange element dynamically sets its state to one bit of the destination address. A value of 0 corresponds to the top output, and 1 corresponds to the bottom output. For example, a message sent by a processor to D_3 (binary address is 011) may move through the exchange elements in the order top, bottom, and bottom outputs. In Figure 2.25, the paths from S0 to D3 and from S5 to D3 are given. Both paths follow the top-bottom-bottom order to reach D3.

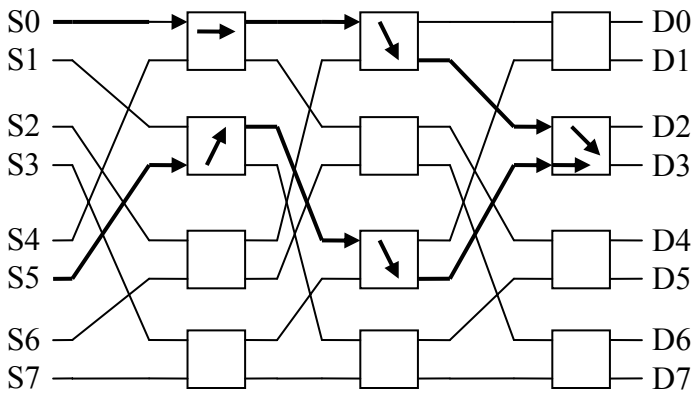


Figure 2.25: Routing on an Omega network with a distributed control scheme.

The diameter of the Omega network is *log N* but only *N/2* exchange elements are actually required. Because all stages of an Omega network are the same, a "recirculating" network can be built to create a single path from source to destination [37]. This is done by constructing a recirculating network designed with a single stage of perfect shuffle*, N/2* exchange elements and *N* registers, and cycling through the network *log N* times.

A major disadvantage of using the Omega network is found when constructing larger Omega networks from existing networks. Constructing a *2N* network from two *N* networks requires rewiring half of the connections in the network to maintain the perfect shuffle interconnection between stages. The rewiring cost is *((N log N)/2 + 2N)*. In Figure 2.26, the extensive rewiring required to construct an 8-Omega network from two 4-Omega networks is illustrated.

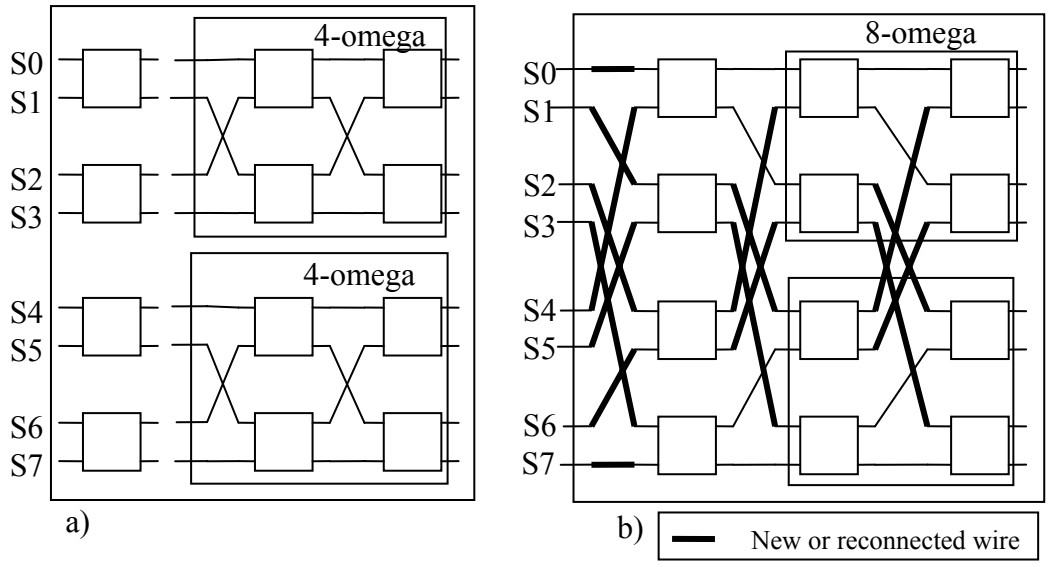


Figure 2.26: Constructing an 8-Omega network with two 4-Omega networks.

The NYU Ultracomputer is a shared memory MIMD that uses an Omega interconnection network to connect processors to memory modules [37]. The network is a static network where the exchange elements of a dynamic network are replaced with special switching processors using a distributed control strategy for routing requests from a processor to a memory module. In addition, the NYU Ultracomputer has a *combining network* where each switching processor contains logic to perform computations to reduce the rate of request to shared memory. For example, if P_3 wants to store a 5 at memory module 2 and simultaneously P_4 wants to store a 10 at memory module 2, the switching processor will combine 5 and 10 and will store a 15 at memory module 2.

2.3.4 Reconfigurable Bus and Mesh

Reconfigurable networks are a classification of dynamic networks that have receive much attention over the last few years. Of particular interest are reconfigurable buses and

reconfigurable meshes [28, 29, 42, 43, 61]. Bondalapati and Prasanna [22, 23] define a reconfigurable bus as a "multi-dimensional array of processing elements connected to a bus through a fixed number of I/O ports. Bus reconfiguration is achieved by locally configuring the switches within each PE." By setting switches, a bus can connect sub-rows (columns) or connect all processors for a global broadcast. Fernandez-Zepeda et al. [29] describe nonlinear, acyclic linear, and cyclic linear reconfigurable bus formations (Figure 2.27). Reconfigurable buses can be shaped to reduce communication time for an algorithm; however branches in nonlinear buses and cycles in cyclic linear buses require more complicated hardware to implement.

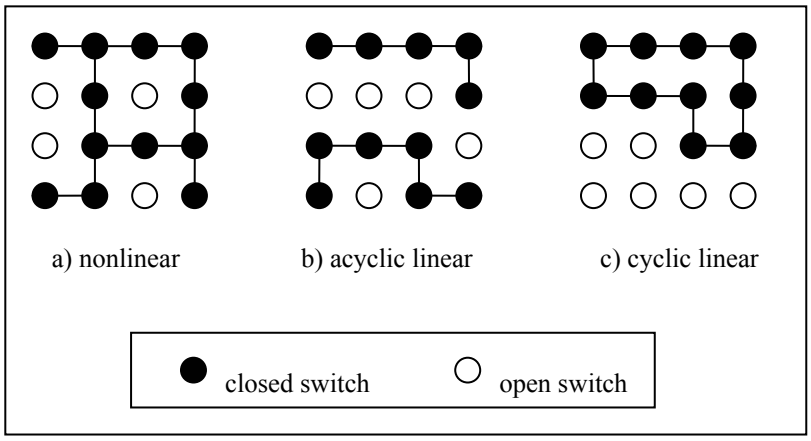


Figure 2.27: Bus shapes

A reconfigurable mesh is an array of processors overlaid with a reconfigurable bus architecture. (Throughout the literature a reconfigurable mesh is considered a "two-dimensional" processor array [22-23], however, there are some algorithms designed for higher dimensional meshes.) Each processor in the mesh has four I/O ports that are typically labeled North, South, East and West to connect with its four neighbors. These ports are locally controlled through switches (i.e. each processor has a switch for each port). In Figure 2.28, various bus

configurations are established on a 3x3 reconfigurable mesh through setting switches. When a node broadcasts a message it can be read in constant time by all nodes connected on the same bus.

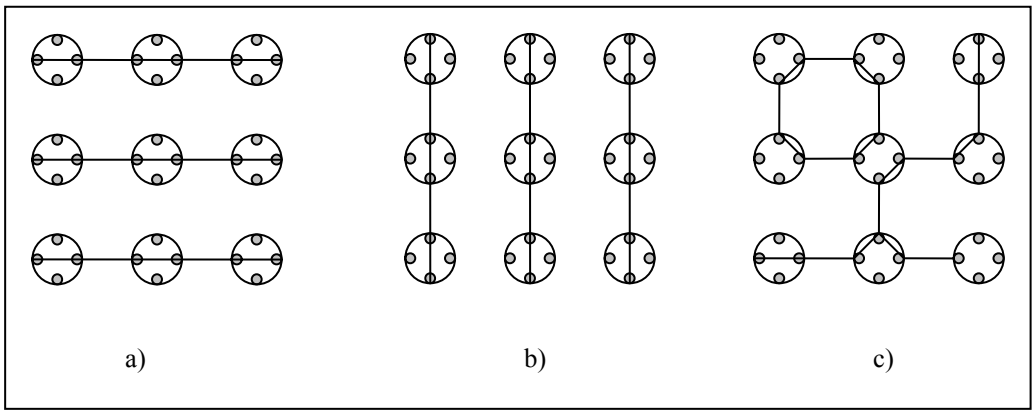


Figure 2.28: Various mesh configurations.

Reconfigurable mesh architectures differ on the following features [22-23]:

- the number of I/O ports open for reading/writing on a single processor at a given configuration.
- the number of connection patterns possible. A *connection pattern* is a single combination of I/O ports (i.e. read from North, write to South or Read from North, write to East etc.).
- global or local controlled connection patterns. If a globally controlled, all processors in the mesh have the same connection pattern for a configuration. If locally controlled, each processor can decide its own connection pattern. In some systems there is conditional global switch settings, where there is a combination of global control and local control.

- partial reconfiguration. With partial reconfiguration a subset of processors have the same connection pattern.
- dynamic reconfiguration. With dynamic reconfiguration a subset of processors can change connection patterns *while* other processors are performing computations.
- size of ALU and memory on each processor in the mesh.

Field Programmable Gate Arrays (FPGAs) are frequently used in reconfigurable meshes. A FPGA is an array of logic blocks overlaid with a network of wires. A typical logic block contains a table look-up, memory and control for logic configuration. "The interconnection network is reconfigured by changing connections between the logic blocks and the wires and by configuration switch boxes which connect different wires."[23] Connections can be modified by downloading external bits of configuration data onto the hardware.

Two models of bus arbitration for reconfigurable meshes include exclusive write and common write. In exclusive-write, one processor broadcasts to bus; in contrast, common-write, multiple processors can broadcast simultaneously as long as all broadcasts have the same value and if its is a single bit [47]. Most meshes use exclusive-write [23]. (Fernandez-Zepeda et al. [29] also describe collision-write in which an error results in the case of multiple simultaneous writes on a bus and collision*-write in which an error only results if different messages are written simultaneously on a bus.)

There are a wide variety of models of reconfigurable meshes. A few include Polymorphic Torus, XC6200 FPGA, RMESH and Content Addressable Array Parallel Processor.

The PARB (Processor Array with Reconfigurable Bus) is described as the "most general purpose" reconfigurable mesh [22]. In the PARB each node has four I/O ports and is connected

in a 2-D mesh. The physical links between two nodes are static so they don't change. In this model, all communication patterns are possible. The Polymorphic Torus is a PARB with torus connections A processor's local internal logic decides the connection pattern. YUPPIE (Yorktown Ultra Parallel Polymorphic Image Engine) is a VLSI implementation of a polymorphic torus that features SIMD, lockstep computation [43, 61]. In this prototype two choices of connection patterns are globally broadcast from a central controller. Internal logic decides which of the two connection patterns to use. A processing element can choose not to participate in a operation by according to data in its local memory. Thus, active status is determined locally. In the YUPPIE, each processing element has a 1-bit ALU that can carry out basic addition and Boolean operations.

The XC6200 FPGA from XILINX is a FPGA that features special logic blocks that contain a 2-to-1 multiplexer. The logic blocks are arranged in 4x4 groups and in 16 X16 tiles. On the XC6200 processors an read and write to the FPGA memory and change the state of the entire FPGA. In addition partial reconfiguration can be accomplished dynamically.

Another example of a reconfigurable mesh is the Content Addressable Array Parallel Processor (CAAPP). CAPP is a 512 x 512 grid of serial bit processors connected with a grid shaped bus locally controlled by of switches.

The *Reconfigurable Mesh* was proposed by Miller et al [47] to capture the "salient features of CAAP and polymorphic-torus." Bondalapati and Prasanna [22-23] describe RMESH as a *NxN* mesh connected to a broadcast *N x N* bus. PEs are connected to a switch at the intersection of the grid lines of the bus. Each bus link between two neighboring PEs has a switch embedded in it that can be control by either of the two neighbors. There are limited connection patterns allowed in the RMESH.

Each node on a mesh can be identified by a (row, column) id. Routing messages between nodes differs according to the type of connection patterns allowed. In Figure 2.29, two options are presented for routing a message from P(0,0) to P(2,2). In the first option, one long bus is established connecting P(0,0) to P(2,2); this can be employed on reconfigurable meshes that allow the nodes to have different connection patterns on a configuration. In the second option all nodes must have the same connection pattern; as a result two different configurations are required. First the nodes connect from West to East to establish horizontal buses, then the nodes connect from North to South to establish vertical buses.

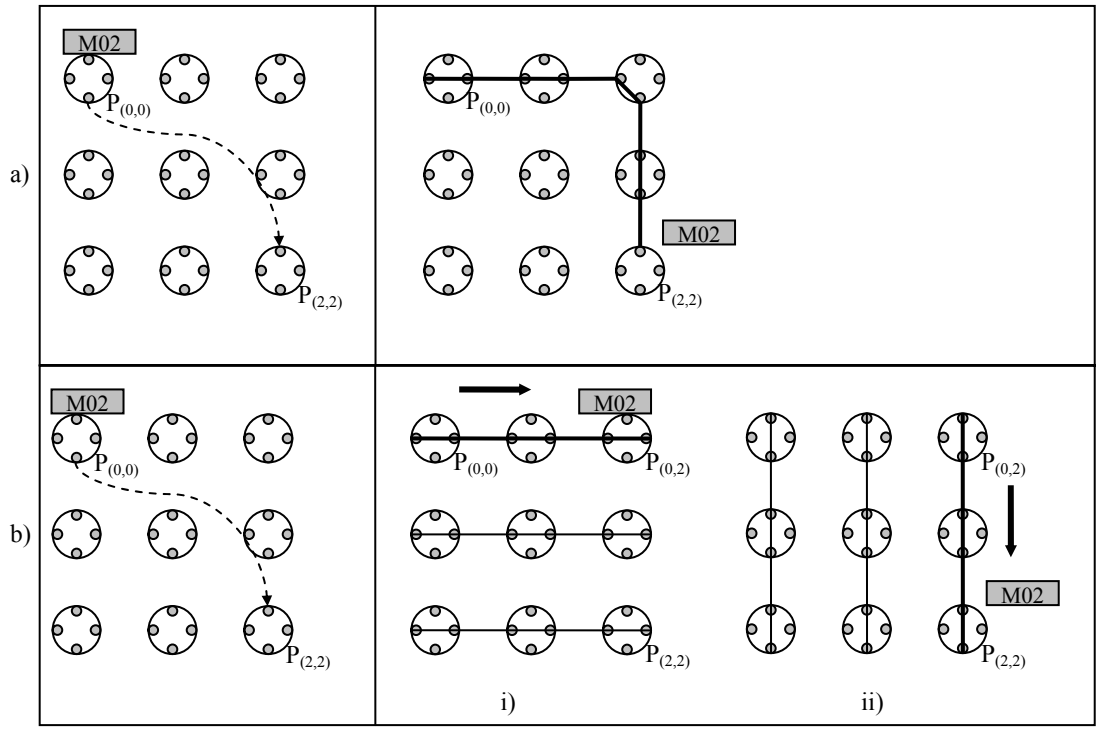


Figure 2.29: Two options for routing a message from $P_{(0,0)}$ to $P_{(2,2)}$

In general, most meshes support global control signals [22-23]. "System-wide synchronization is needed for harmonious reconfiguration of the system" [42]. Even though dynamic reconfiguration is possible in theory, implementing dynamic change of configuration

for a subset of processors while others are performing computations is considered very expensive. Furthermore, reconfigurable meshes are characterized by unpredictable signal delays due to long path and number of processors connected to wire carrying signal [22-23]. In some cases "clock timing are based on the worst-case shortest path which degrades performance overall for the system".

Most reconfigurable meshes allow only one processor on a bus to write to the bus at a time; having exclusive access to bus resources limits throughput [66]. Lidstone et al [42] report that as the number of processors increase, the message passing latency also increases due to delays encountered in regulating data across the network. There have been several articles published reporting constant time algorithms for implementation on reconfigurable meshes [43,61]; however the meshes may require a higher dimension than 2-D. As Lin et al. [44] reports, "even though a constant time algorithm for matrix multiplication was reported, as many as N^4 processors are used" to solve *NxN* matrix multiplication.

Clearly there is still room for additional reconfigurable architectures. The MultiRing presented in this dissertation is a reconfigurable network that features organizing processors into different collection of rings.

2.3.5 Summary of Dynamic Networks

In this section, bus, crossbar and multi-stage dynamic interconnection networks have been presented. In dynamic networks, switches are used to connect processors indirectly. One definition of efficient communication on a parallel system is that it allows concurrent communication between multiple pairs of processors with minimal communication latency.

A bus provides minimal communication latency between processors since a single switch is used to connect a processor to the bus; however only one processor can write to the bus at a given time. A crossbar provides both minimal communication latency and allows multiple pairs of processors to communicate simultaneously. The disadvantage of using crossbars is that many switches must be added when expanding to allow additional processors. The goal of a MIN is to allow *N* messages to be sent simultaneously without requiring the N^2 switches as an *NxN* crossbar switch. The cost of using an MIN is the increase in communication latency as messages are routed through a series of switching elements.

2.4 Concluding Remarks

In this chapter, we discussed some basic parallel programming definitions and provided examples of existing static and dynamic networks. Static networks connect processors directly using fixed links. They are best used with programs that have a fixed communication pattern that mimics the layout of the network. Table 2.3 summarizes the comparison of the different static networks based on degree, diameter, bisection width and number of links.

Table 2.3: Comparison of Static Networks

Topology	Degree	Diameter	Bisection	Number of Links
Linear Array	2	N-1	1	N-1
Ring	2	N/2	2	N
2-dimensional Grid	4	$2(N^{1/2}-1)$	$N^{1/2}$	$2(N-N^{1/2})$
Hypercube	log N	log N	N/2	(N/2)*(log N)
Completely Connected	N-1	1	$(N/2)^2$	N(N-1)/2

The completely connected networks are only used if there are a small number of processors to connect or if hardware cost is not a factor. In general, the lower dimensional networks are easier to lay out because they involve mostly local connects and fewer wires. The hypercube networks were initially popular because parallel algorithms designed for other network topologies could be implemented on the hypercube without many changes. However, expanding the hypercube to allow additional processors requires changing the node degree and makes manufacturing hypercubes complicated.

In dynamic networks, the node degree remains constant and processors communicate through switches. The number of communication ports for each node as new processors are added to the network is not increased; instead, the number of switches is increased. Either a bus, with a simple design and high potential for contention, or a crossbar network, with low contention and complex design, is used if there are only a few processors.

For applications that involve a lot of processors and require a flexible communication pattern, a multistage interconnection network is used. Communication between pairs of processors is established through a grid of switch elements. If a distributed control strategy is used, a dynamic network can convert into a direct network by replacing every switch element with a processor that is dedicated to routing transmissions. Table 2.3 summarizes the comparison of the different dynamic networks based on diameter, number of switches, concurrency and rewiring costs.

Table 2.4: Comparison of Dynamic Networks[2]

	Diameter	Number of Switches	Concurrency	Rewiring cost
N –Bus	1	N	1	N
NXN Crossbar	1	N^2	N	N^2
N-Butterfly	log N	2N log N	N	2N
N-Omega	log N	2N log N	N	(N log N)/2 + 2N
N-Benes	2 log N -1	2N(2log N-1)	N	4N

The main issue between static and dynamic networks has to do with increasing either communication ports or switches. A high node degree in static networks can lead to complex multiplexing strategies for the numerous ports for each node and also to high rewiring expansion costs. In a dynamic network, performance is hindered by the fall-through latency needed to navigate through the collection of switches.

It has become clear that to be useful in an ever-changing environment, a parallel system must be able to expand easily so that it can accommodate additional processors while maintaining efficient communication between processors. The MultiRing is designed to meet those two needs. The MultiRing is scalable since all nodes connect to a MultiRing switch with only two links. Inside the MultiRing switch is a dynamic interconnection network of switching elements that grows as processors are added.

To reduce conflicts, most MINs employ a distributed control strategy where smart switches are employed to determine a message's communication path by its destination address. These smart switches are more expensive than simple switches. To build an inexpensive

[2] Assumes 2x2 exchange elements with 4 switches for multistage interconnection networks.

network, simple switches that are globally controlled are used. The communication paths are limited from the onset since only certain reconfigurations of the switch are allowed. These configurations form multiple rings which allow pairs of processor to communicate simultaneously.

In the next chapter, the use of existing MINs to provide the configurations needed for the MultiRing switch is explored.

CHAPTER 3 : MAPPING THE MULTIRING ON EXISTING INTERCONNECTION NETWORKS

Chapter 2 examined existing static and dynamic interconnection networks and established that a parallel system should have both efficient processor communication and a limited number of physical links. The MultiRing is a reconfigurable network that should be mapped onto a network with a low diameter and node degree. Dynamic networks, in particular, the multi-stage interconnection networks, allow multiple messages to be sent in parallel and have a set degree of connectivity. However, not all MINS allow all permutations of the network. The Benes network allows all permutations but its control strategy is complicated and its communication latency is twice that of the Butterfly and Omega networks. This chapter explores using the Butterfly and Omega networks to create the MultiRing switch.

A MultiRing *maps* onto a MIN by setting the exchange elements to establish the required communication of each configuration of the MultiRing. The MultiRing configurations form multiple independent rings which are designed to allow pairs of processors to communicate simultaneously while avoiding conflicts. In an 8-node MultiRing, there are four configurations possible: 8 rings of 1 node, 4 rings of 2 nodes, 2 rings of 4 nodes and 1 ring of 8 nodes. See Figure 3.1 for the four configurations of 8-node MultiRing with nodes labeled *A –H*.

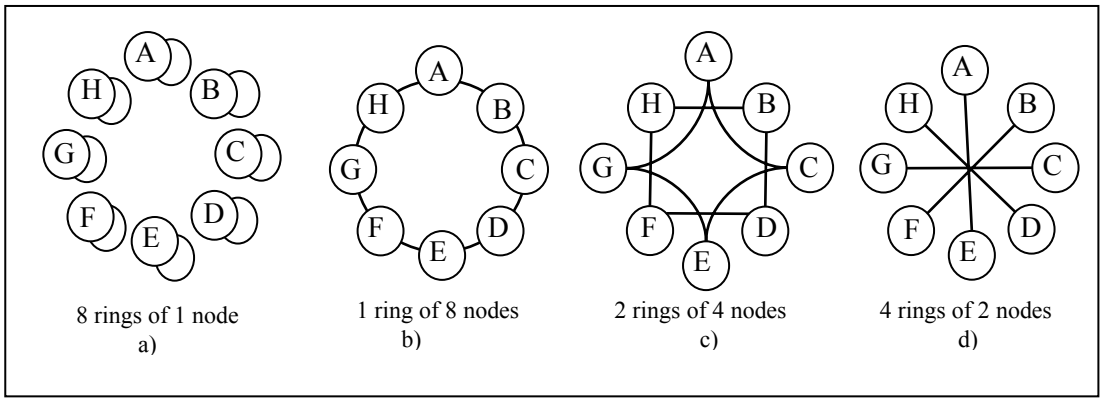


Figure 3.1: Four configurations of an 8-node MultiRing.

In section 3.1 an 8-node MultiRing is mapped onto a Butterfly, and in section 3.2, it is mapped on to an Omega network. Smart switching elements that independently determine the path of a message according to its destination are expensive, so in order to build an inexpensive switch, simple switching elements are used that are controlled by a single control bit. Both of the networks use 2x2 exchange elements and a synchronized control strategy for setting the states of the exchange elements. All exchange elements change in lockstep to establish a fixed communication path.

The single control bit is used to set an exchange element to either bar or cross state. In the worst case, controlling the state of all exchange elements requires maintaining a different control bit for each exchange element. However, since a MultiRing with $N=2^r$ nodes has only *log N* different configurations of the network, a subset of the exchange elements share the same control. A global control word with $r(r+1)/2$ bits controls the Butterfly and Omega networks. 8-Butterfly and 8-Omega networks are required to map the 8-MultiRing, thus requiring a control word with 6 bits (i.e. 3(3+1)/2). The Butterfly and Omega have different rewiring costs for creating 2*N* networks with two *N*-networks and placing the control bits.

3.1 Mapping the MultiRing on the Butterfly Network

An 8-Butterfly network (Chapter 2) has a collection of exchange elements organized in *4* rows and *3* columns which are connected using the butterfly interconnection pattern. Contrary to expectations, the MultiRing *does not* map directly onto the Butterfly network. In the configuration with eight rings of 1 node, each node should communicate with itself (i.e., there should be a path through the exchange elements from source *A* to destination *A*). In attempts to establish this configuration conflicts occur with the state of the exchange elements (Figure 3.2). The shaded elements with both bar and cross states depicted represent conflicting control signals. If node *A* communicates with itself, all of the exchange elements in the first row are set to bar states. However, node *B* will not be able to connect with itself; it requires the first exchange element in the first row to be set to cross state.

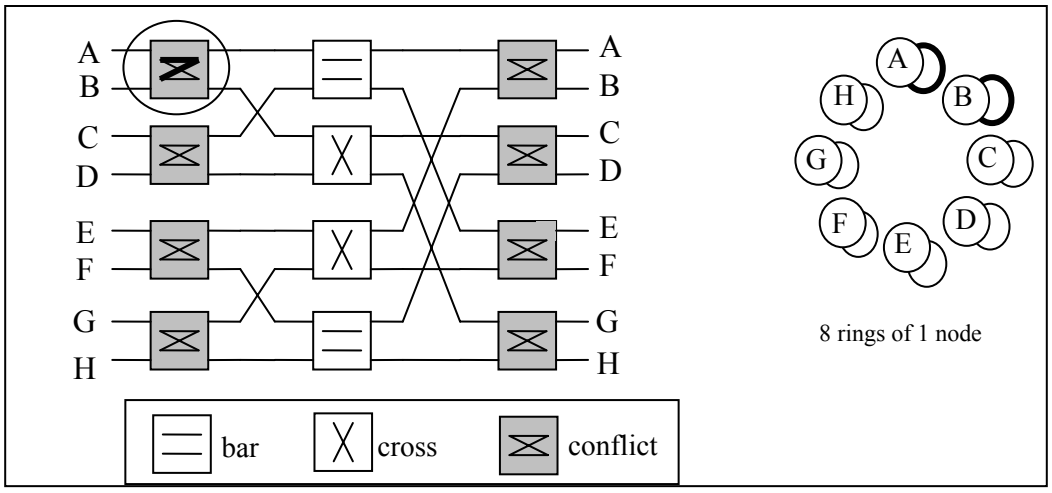


Figure 3.2: Conflicts when mapping MultiRing on an 8-Butterfly.

He and Arabnia [30] addressed this problem by modifying the Butterfly network. The network they designed will be referred to as an HAB network. An *N*-HAB connects *N* nodes. Additional links were connected to the output of the network using an inverse-perfect shuffle so

that setting all exchange elements to bar state establishes the communication necessary for rings of only one node(Figure 3.3).

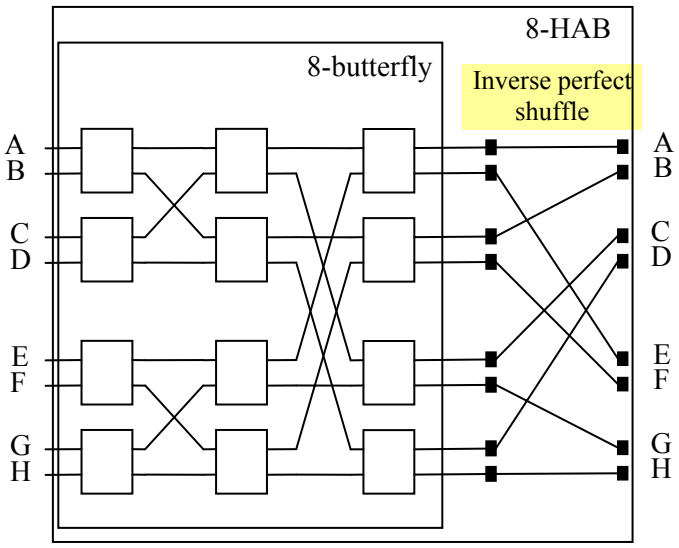


Figure 3.3: 8-HAB network.

In Figure 3.4 - Figure 3.7, the four different configurations of the 8-node MultiRing are mapped onto an 8-HAB network (assuming clockwise transmission on the ring only). In Figure 3.4, the paths from *A* to *A* and *B* to *B* on the configuration with 8 rings of one node are highlighted. In Figure 3.5, the paths from *A* to *B* and from *B* to *C* are highlighted on the ring of 8 nodes. In Figure 3.6, the paths from *A* to *C* and from *C* to *E* are highlighted in the configuration with 2 rings of 4 nodes. In Figure 3.7, the paths from *A* to *E* and from *E* to *A* are highlighted in the configuration with 4 rings of 2 nodes.

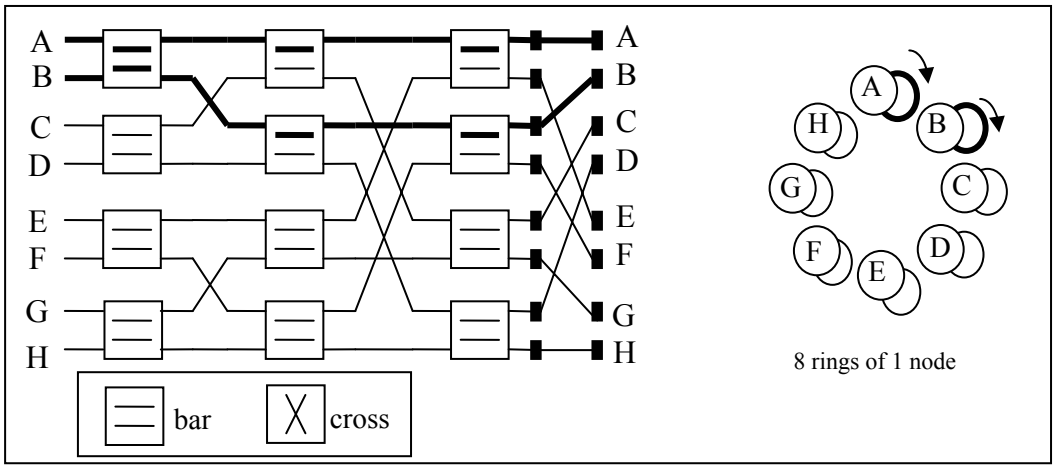


Figure 3.4: 8 rings of 1 node on 8-HAB network.

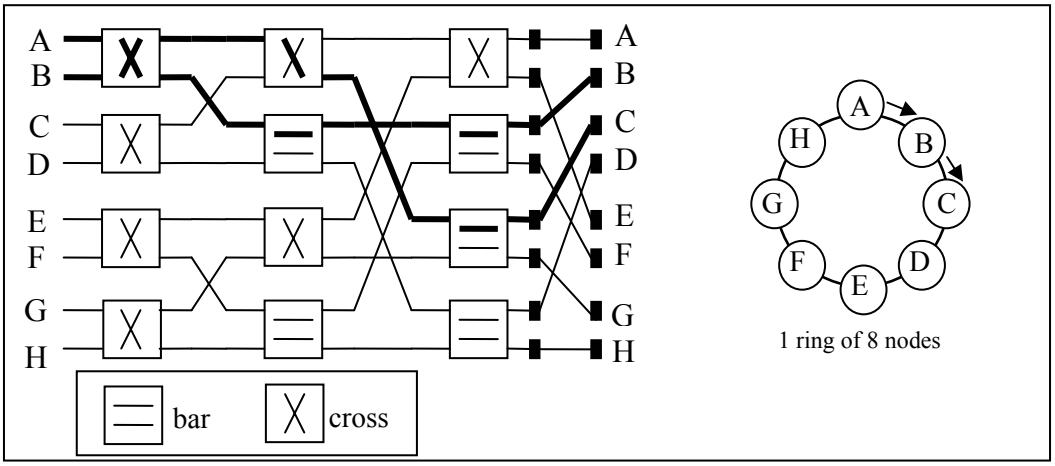


Figure 3.5: Ring of 8 nodes on 8-HAB network.

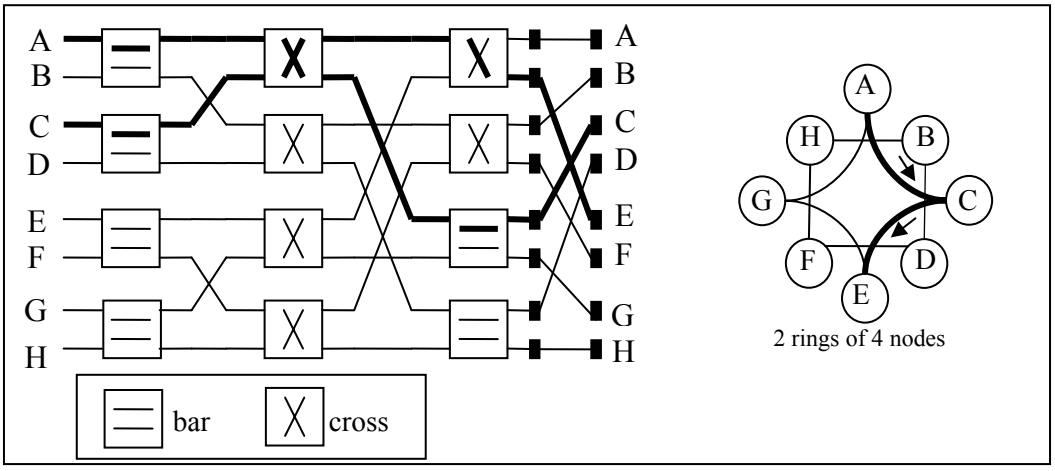


Figure 3.6: 2 rings of 4 nodes on 8-HAB network.

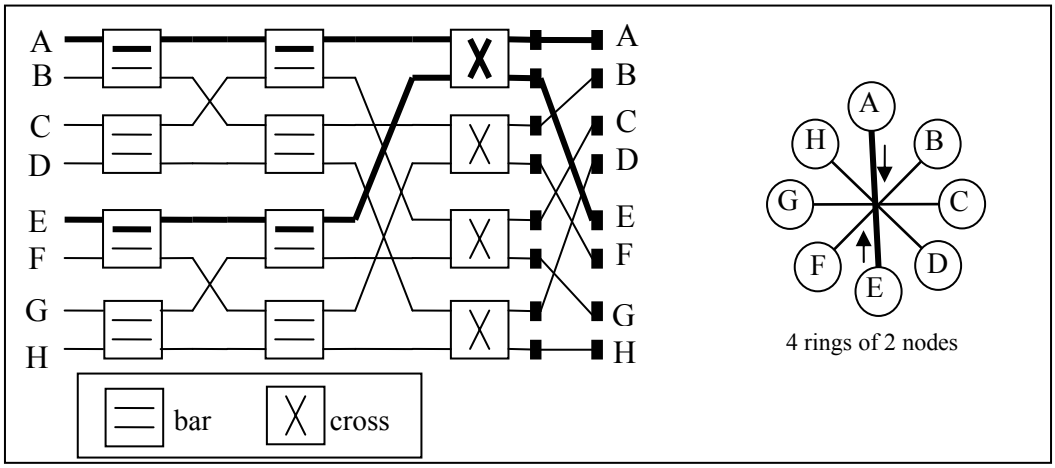


Figure 3.7: 4 rings of 2 nodes on 8-HAB network.

The HAB network scales modularly. In order to create a 2*N*- HAB network with two *N*-Butterfly networks, it is necessary to construct a 2*N*- Butterfly network and use inverse perfect shuffle to reconnect the output. Construction of an 8-HAB network from two 4-Butterfly networks is illustrated in Figure 3.8.

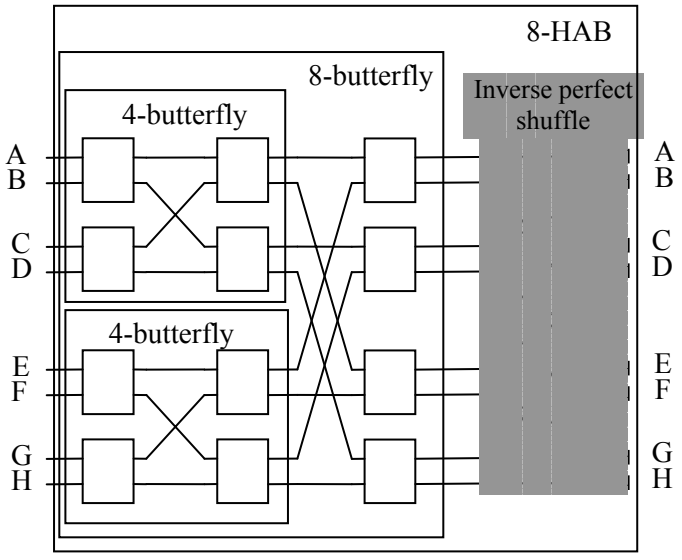


Figure 3.8 Constructing an 8-HAB network from two 4-Butterfly networks.

Managing the HAB network is complex and requires different control signals. An N-HAB with 2^r nodes contains r columns of exchange elements, numbered from 0 to $r-1$ and 2^{r-1} rows. A global control word comprised of several different control bits manages the state of the network and establishes the communication patterns required for a configuration of the ring. Each exchange element is controlled by one bit of the control word. However, a subset of exchange elements in the same column, share the same control bit. One control bit is needed to set the exchange elements in column 0. Two different control bits are needed to set the exchange elements in column 1. Three different bits control the state of elements in column 2. This process repeats, until in column $r-1$, r control bits are required.

In mapping a MultiRing on a HAB network, the control word is set according to the number of rings in a configuration. In a MultiRing with $N=2^r$ nodes, the number of rings R can be expressed in binary in $r+1$ bits, $R= R_rR_{r-1}\ldots R_2R_1R_0$. In an 8-node MultiRing, there can be 8 rings, 1 ring, 2 rings, or 4 rings which are 0000, 0001, 0010, and 0100 respectively in binary. "8 rings" describes the configuration in which each node connects with itself, thereby creating rings of 1 node.

The notation C_{ij} represents a single control bit, where i is a column number from 0 to $r-1$ and j represents a different control for that column, and where $0 \leq j \leq i$. (i.e., C_{ij} is the $(j+1)$th control bit for column i.) Column i has $i+1$ different control bits: $C_{i0}, C_{i1}, C_{i2}, \ldots, C_{ii}$. The control signals in the 8-HAB are labeled $C_{00}, C_{10}, C_{11}, C_{20}, C_{21}$, and C_{22} (Figure 3.9). C_{00} controls the exchange elements in column 0, C_{10} and C_{11} control the exchange bits in column 1; and C_{20}, C_{21}, and C_{22} control the exchange elements in column 2.

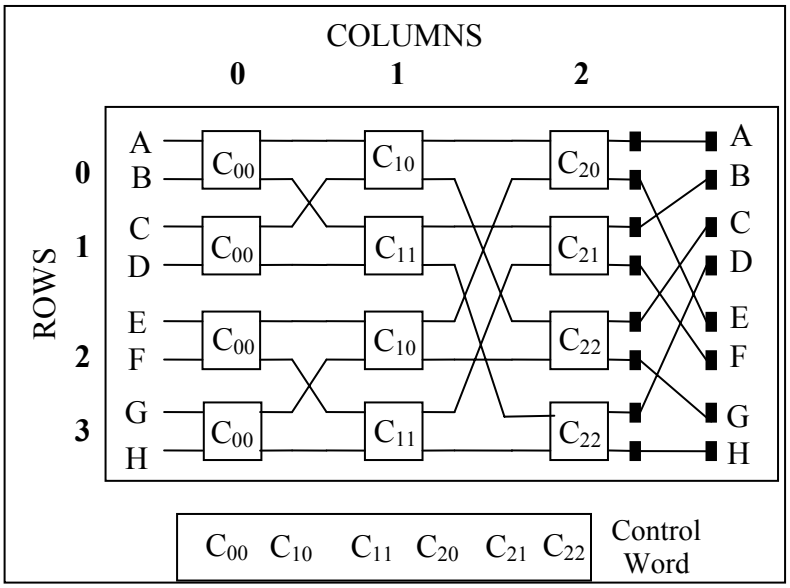


Figure 3.9: Location of controls for 8-HAB.

He and Arabnia [30] set control bit C_{ij} is set as follows:

$$C_{ij} = \begin{cases} R_i & \text{for } i = j \\ R_j \text{ OR } R_{j+1} \text{ OR } R_{j+2} \text{ OR } \ldots \text{ OR } R_i & \text{for } i > j \end{cases}$$

This can be expressed recursively as:

$$C_{ij} = \begin{cases} R_i & \text{for } i = j \\ C_{i-1,j} \text{ OR } R_i & \text{for } i > j \end{cases}$$

In total, $B = r(r+1)/2$ (i.e., $1 + 2 + 3 + \ldots + r$) different control bits are needed in a control word. Thus, in an 8-HAB, six different control bits are required. The bth control bit in the control word where $0 \le b < B$ is C_{ij} where the formulas[3] for i and j are:

$$i = \left\lfloor \frac{\sqrt{(1+8n)}-1}{2} \right\rfloor \quad \text{and} \quad j = b - \frac{T_b \cdot (T_b - 1)}{2} \quad \text{where} \quad T_b = \left\lfloor \frac{1+\sqrt{1+8n}}{2} \right\rfloor .$$

[3] The pattern for *i (0,1,1,2,2,2,3,3,3,3,...)* and *j (0,0,1,0,1,2,0,1,2,3,...)* were entered into an online encyclopedia recommended by John Robertson: Reference: On-Line Encyclopedia of Integer Sequences for the corresponding formula. < http://www.research.att.com/~njas/sequences/ >

The exchange element E_{st} in row s for $0 \leq s < 2^{r-1}$ and column t for $0 \leq t < r$ is controlled by C_{tj} where

$$j = \begin{cases} 0 & \text{If } ((s\%2^t) == 0) \\ \lfloor \log(s\%2^t) + 1 \rfloor & \text{Else} \end{cases}$$

The controls for an 8-HAB establishing the different MultiRing configurations are in Figure 3.10. Also, the value of all exchange elements' control bit is provided. The 8 rings of 1 node control settings correspond to Figure 3.4. The 1 ring of 8 nodes control settings correspond to Figure 3.5. The 2 rings of 4 nodes control settings correspond to Figure 3.18. The 4 ring of 2 nodes settings correspond to Figure 3.7.

Number of Rings	R	Controls for 8-HAB					
	$R_3R_2R_1R_0$	C_{00}	C_{10}	C_{11}	C_{20}	C_{21}	C_{22}
8	1 0 0 0	0	0	0	0	0	0
1	0 0 0 1	1	1	0	1	0	0
2	0 0 1 0	0	1	1	1	1	0
4	0 1 0 0	0	0	0	1	1	1

8 Rings of 1 node		
C_{00} 0	C_{10} 0	C_{20} 0
C_{00} 0	C_{11} 0	C_{21} 0
C_{00} 0	C_{10} 0	C_{22} 0
C_{00} 0	C_{11} 0	C_{22} 0

1 Ring of 8 nodes		
C_{00} 1	C_{10} 1	C_{20} 1
C_{00} 1	C_{11} 0	C_{21} 0
C_{00} 1	C_{10} 1	C_{22} 0
C_{00} 1	C_{11} 0	C_{22} 0

2 Rings of 4 nodes		
C_{00} 0	C_{10} 1	C_{20} 1
C_{00} 0	C_{11} 1	C_{21} 1
C_{00} 0	C_{10} 1	C_{22} 0
C_{00} 0	C_{11} 1	C_{22} 0

4 Rings of 2 nodes		
C_{00} 0	C_{10} 0	C_{20} 1
C_{00} 0	C_{11} 0	C_{21} 1
C_{00} 0	C_{10} 0	C_{22} 1
C_{00} 0	C_{11} 0	C_{22} 1

0	bar state

1	cross state

Figure 3.10: Controls for 8-HAB for each MultiRing configuration.

The controls for a 16-HAB for each configuration in a 16-MultiRing are provided in Figure 3.11. Notice that 8-HAB controls are included in the 16-HAB. The locations of controls

for the 8-HAB switching elements are identified with a dark square and the number 8 in the center. When the 16-HAB is set for either 16 or 8 rings, the 8-HAB controls are set for 8 rings. When the 16-HAB is set for 2 rings the 8-HAB controls are set for 2 rings. When the 16-HAB is set for 4 rings, the 8-HAB controls are set for 4 rings.

Number of Rings	R	Controls for 16-HAB								
	$R_4R_3R_2R_1R_0$	C_{00}	C_{10}	C_{11}	C_{20}	C_{21}	C_{22}	C_{30}	C_{31}	C_{32}	C_{33}
16	1 0 0 0 0	0	0	0	0	0	0	0	0	0	0
1	0 0 0 0 1	1	1	0	1	0	0	1	0	0	0
2	0 0 0 1 0	0	1	1	1	1	0	1	1	0	0
4	0 0 1 0 0	0	0	0	1	1	1	1	1	1	0
8	0 1 0 0 0	0	0	0	0	0	0	1	1	1	1

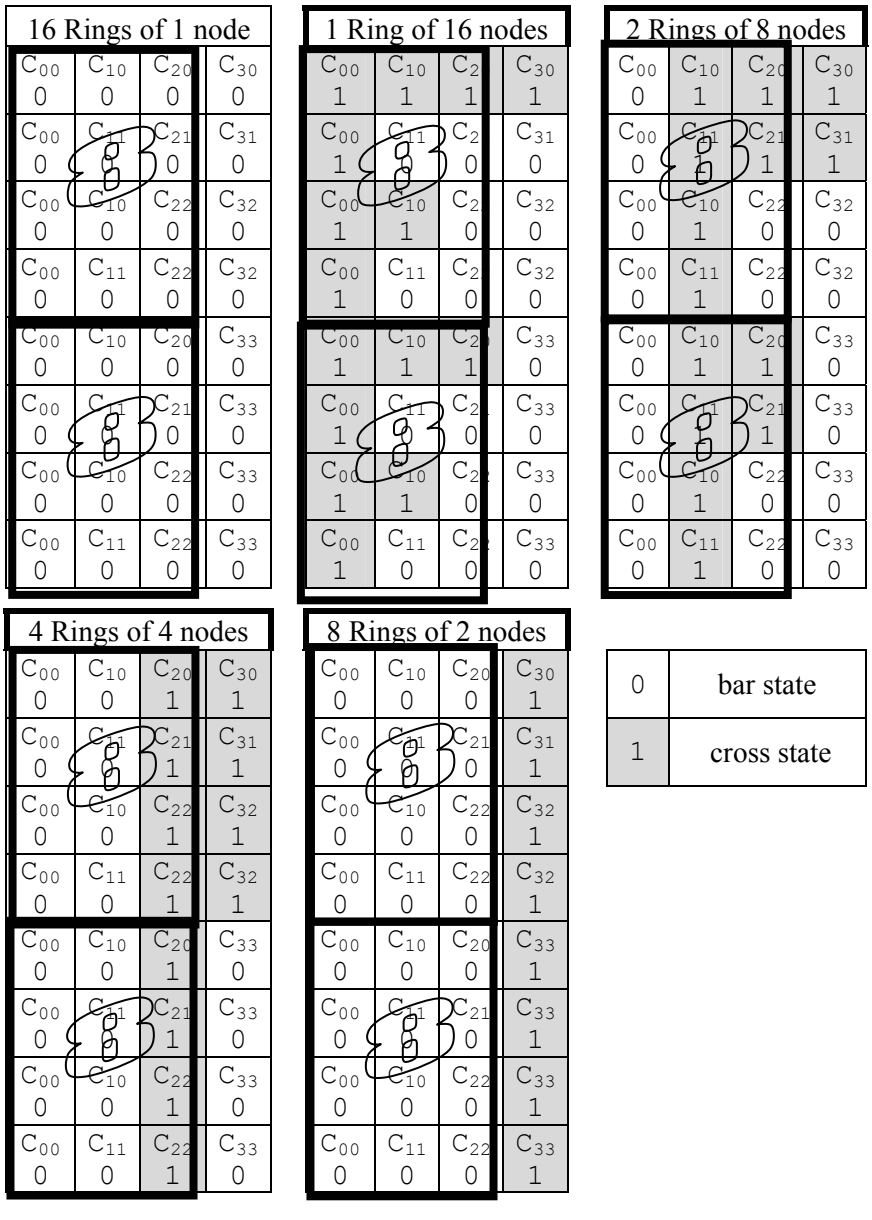


Figure 3.11: Controls for 16-HAB for each MultiRing configuration.

When creating a 2*N*-HAB with two existing *N*-HAB networks, the existing control word remains connected to the *N*-HAB networks without modifying any connections, and *r+1* new control signals are added to the control word. An additional column of exchange elements is managed by the new controls. (See Figure 3.12)

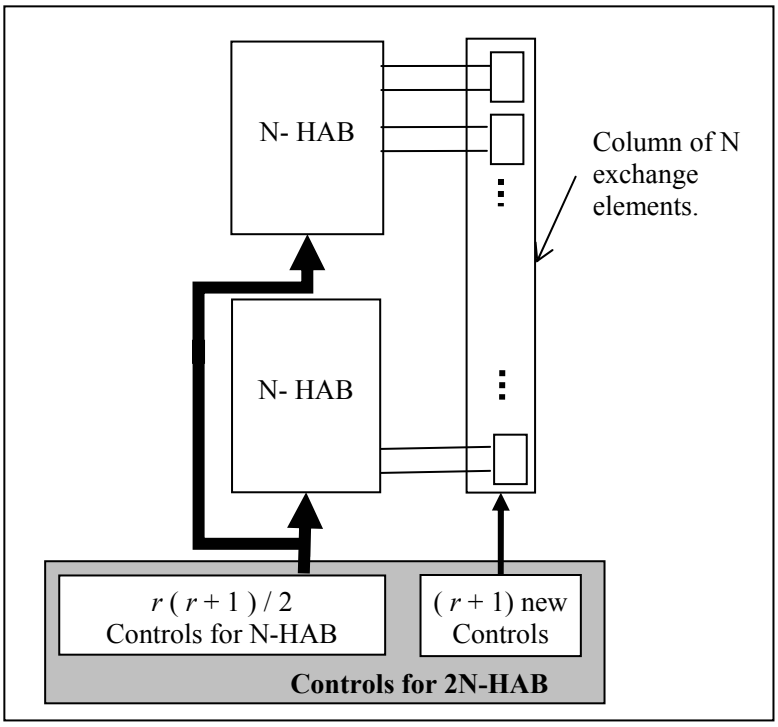


Figure 3.12: Controls for 2N-HAB

As an additional illustration, in Figure 3.13, the controls for a 16-HAB are built recursively from smaller HAB networks. The controls for a 2-HAB are used in a 4-HAB. The controls for a 4-HAB are used in an 8-HAB. Finally, the controls for the 8-HAB are used in the 16-HAB.

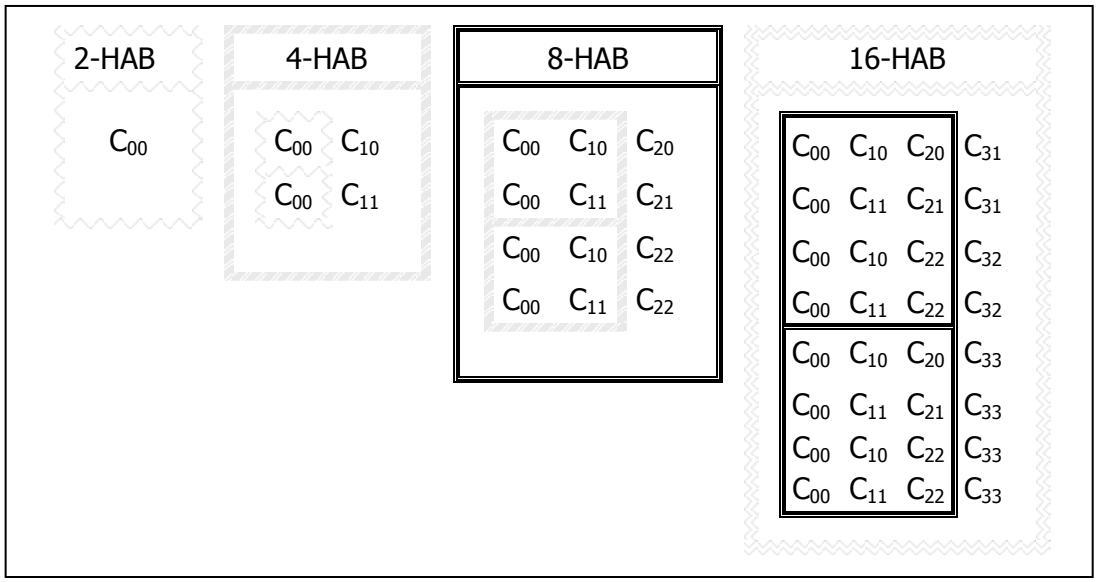


Figure 3.13: 16-HAB is recursively built from smaller networks.

All of the rings previously examined were clockwise. If counter-clockwise rings were required, the exchange elements would be set differently for 2 rings of 4 nodes (Figure 3.14) and 1 ring of 8 nodes (Figure 3.15). As observed by He and Arabnia [31], an additional control bit is needed to allow both clockwise and counter-clockwise configurations. Providing counter-clockwise rings adds to the complexity of creating the control word and determining the control bit for a specific exchange element.

The shaded exchange elements in Figure 3.14 and Figure 3.15 are set to a different state to enable clockwise communication. In Figure 3.14, the paths in the counter-clockwise ring of 4 nodes from *A* to *G* and from *C* to *A* are highlighted. In Figure 3.15, the paths in the counter-clockwise ring of 8 nodes from *A* to *H* and from *B* to *A* are highlighted.

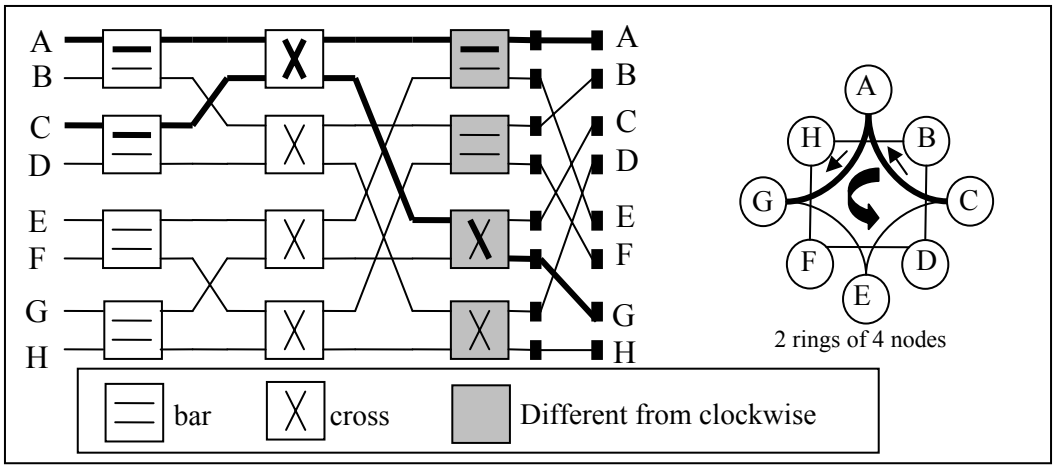


Figure 3.14: Two counter-clockwise rings of 4 nodes on 8-HAB.

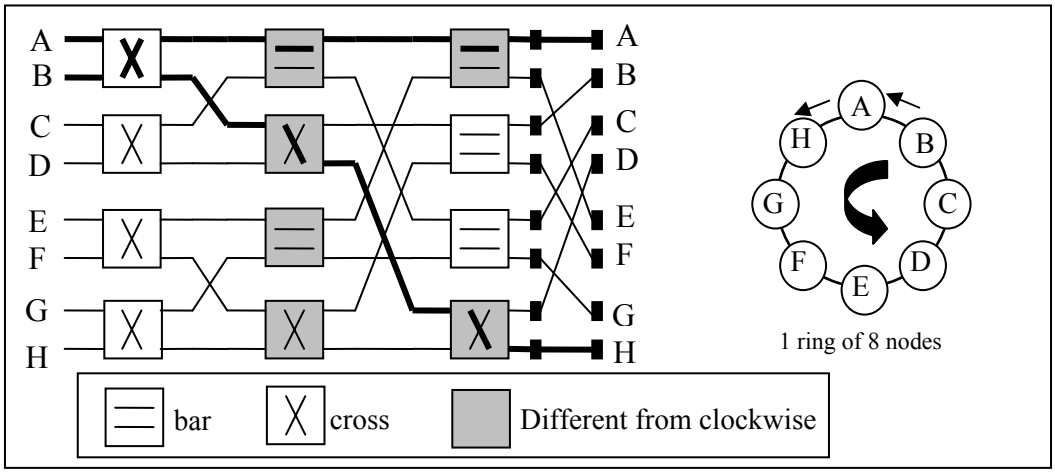


Figure 3.15: Counter-clockwise ring of 8 nodes on 8-HAB.

3.2 Mapping the MultiRing on the Omega Network

An 8-Omega network as described in Chapter 2 has a collection of exchange elements organized in *4* rows and *3* columns which are connected using a series of perfect shuffles. An 8-MultiRing maps directly onto the Omega network without conflicts. Figure 3.16- Figure 3.19 illustrate the states of the exchange elements for each of the four configurations required for an

8- MultiRing (assuming clockwise flow of data). In Figure 3.16, the paths from *A* to *A* and *B* to *B* on the configuration with 8 rings of one node are highlighted. In Figure 3.17, on the configuration of 1 ring of 8 nodes, the paths from *A* to *B* and from *B* to *C* are highlighted. In Figure 3.18, the paths from *A* to *C* and from *C* to *E* in the configuration with 2 rings of 4 nodes are highlighted. In Figure 3.19, the paths from *A* to *E* and from *E* to *A* in the configuration with 4 rings of 2 nodes are highlighted.

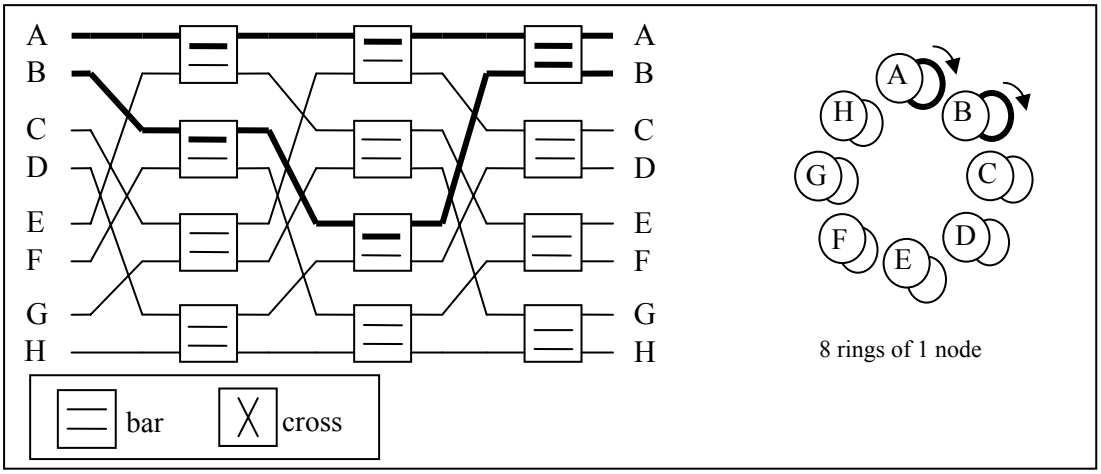


Figure 3.16: 8 rings of 1 node on 8-Omega network.

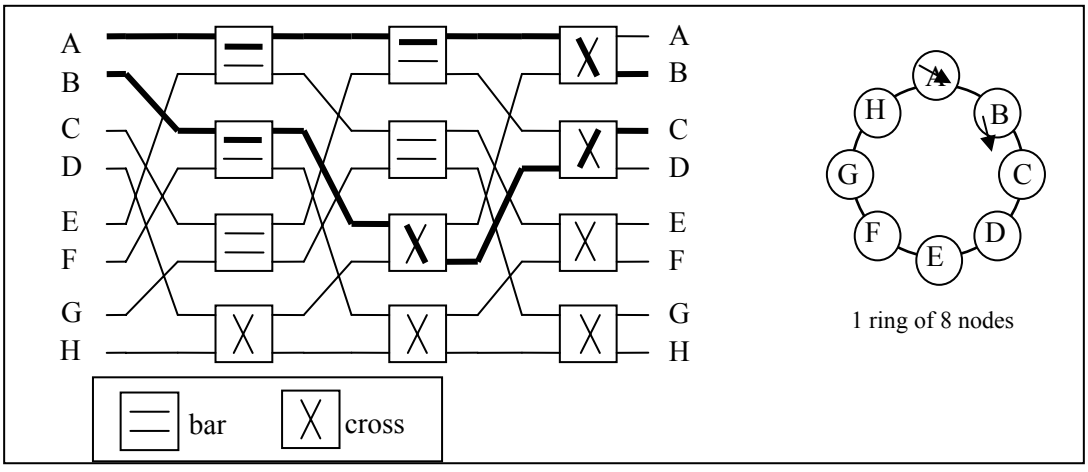


Figure 3.17: Ring of 8 nodes on 8-Omega network

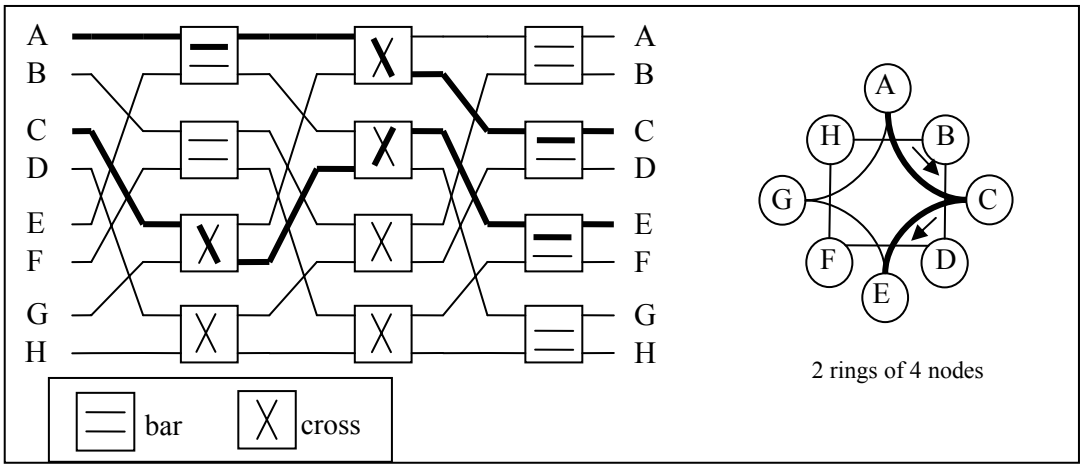


Figure 3.18: 2 rings of 4 nodes on 8-Omega network

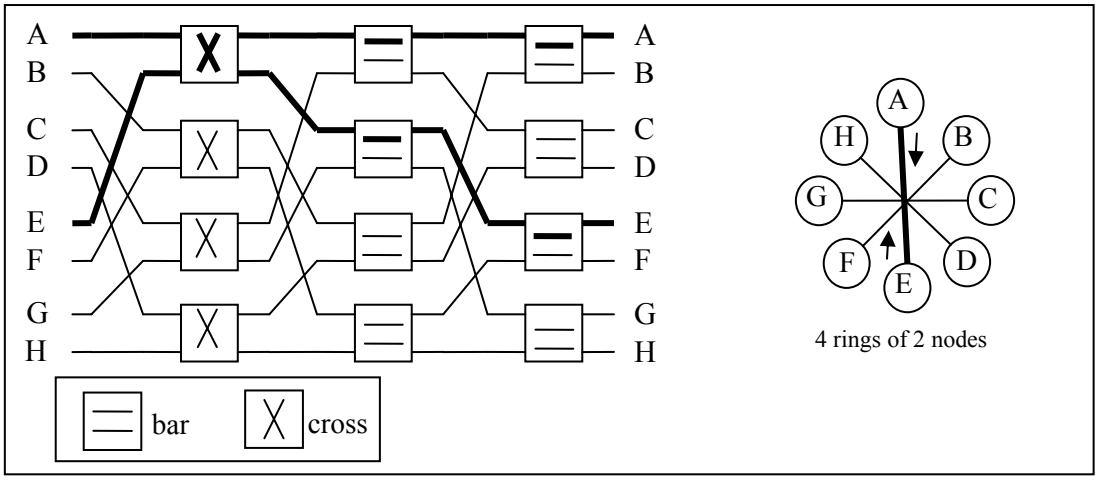


Figure 3.19: 4 rings of 2 nodes on 8-Omega network.

Even though it is possible to map the MultiRing network on an Omega network, scalability is a major disadvantage. As discussed in Chapter 2, constructing a 2*N*-Omega network from two existing *N*-Omega networks requires extensive rewiring. This rewiring also affects connections with the control signals.

As in the previous section, the notation C_{ij} represents a single control bit, where *i* is a column number from *0* to *r-1* and *j* represents a different control for that column, where $0 \leq j \leq$

i. As with the HAB, an Omega network has six different control signals which are labeled C_{00}, C_{10}, C_{11}, C_{20}, C_{21}, and C_{22}. In contrast to the HAB, on the Omega network, the columns are numbered from right to left and the rows are numbered from bottom to top. In Figure 3.20, C_{00} controls the exchange elements in column 0, C_{10} and C_{11} control the exchange bits in column 1; and C_{20}, C_{21}, and C_{22} control the exchange elements in column 2.

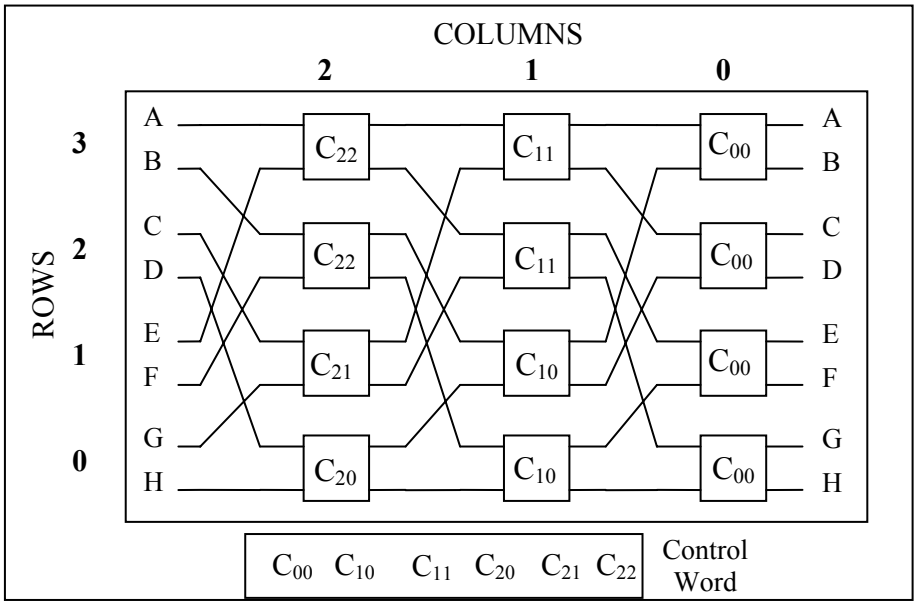


Figure 3.20: Location of controls for 8-Omega network

The Omega network uses the same control word created for the HAB network. Control signals are based on the number of rings in a given configuration; however, the control bits manage *different* groups of exchange elements than in the HAB. The exchange element E_{st} in row s and column t is controlled by C_{tj} where

$$j = \left\lceil \log\left(s2^{t+1-r} + 1 \right) \right\rceil$$

The controls for an 8-Omega and value of the control bit for each exchange element are provided for each configuration of the MultiRing in Figure 3.21. The 8 rings of 1 node settings

correspond to Figure 3.16. The 1 ring 8 nodes control settings correspond to Figure 3.17. The 2 rings of 4 nodes control settings correspond to Figure 3.18. The 4 rings of 4 nodes control settings correspond to Figure 3.19.

Number of Rings	R	Controls for 8-Omega					
	$R_3R_2R_1R_0$	C_{00}	C_{10}	C_{11}	C_{20}	C_{21}	C_{22}
8	1 0 0 0	0	0	0	0	0	0
1	0 0 0 1	1	1	0	1	0	0
2	0 0 1 0	0	1	1	1	1	0
4	0 1 0 0	0	0	0	1	1	1

8 Rings of 1 node		
C_{22} 0	C_{11} 0	C_{00} 0
C_{22} 0	C_{11} 0	C_{00} 0
C_{21} 0	C_{10} 0	C_{00} 0
C_{20} 0	C_{10} 0	C_{00} 0

1 Ring of 8 nodes		
C_{22} 0	C_{11} 0	C_{00} 1
C_{22} 0	C_{11} 0	C_{00} 1
C_{21} 0	C_{10} 1	C_{00} 1
C_{20} 1	C_{10} 1	C_{00} 1

2 Rings of 4 nodes		
C_{22} 0	C_{11} 1	C_{00} 0
C_{22} 0	C_{11} 1	C_{00} 0
C_{21} 1	C_{10} 1	C_{00} 0
C_{20} 1	C_{10} 1	C_{00} 0

4 Rings of 2 nodes		
C_{22} 1	C_{11} 0	C_{00} 0
C_{22} 1	C_{11} 0	C_{00} 0
C_{21} 1	C_{10} 0	C_{00} 0
C_{20} 1	C_{10} 0	C_{00} 0

0	bar state

1	cross state

Figure 3.21: Controls for 8-Omega for each MultiRing configuration

The controls for a 16-Omega and the value of the control bits for all exchange elements are in Figure 3.22. Notice there is no symmetry with the previous 8-Omega exchange elements in configurations for 1 and 2 rings, even though the same control bits in the 8-Omega control word are used in the 16-Omega control word.

Number of Rings	R	Controls for 16-Omega									
	$R_4R_3R_2R_1R_0$	C_{00}	C_{10}	C_{11}	C_{20}	C_{21}	C_{22}	C_{30}	C_{31}	C_{32}	C_{33}
16	0 0 0 0	0	0	0	0	0	0	0	0	0	0
1	0 0 0 1	1	1	0	1	0	0	1	0	0	0
2	0 0 1 0	0	1	1	1	1	0	1	1	0	0
4	0 1 0 0	0	0	0	1	1	1	1	1	1	0
8	1 0 0 0	0	0	0	0	0	0	1	1	1	1

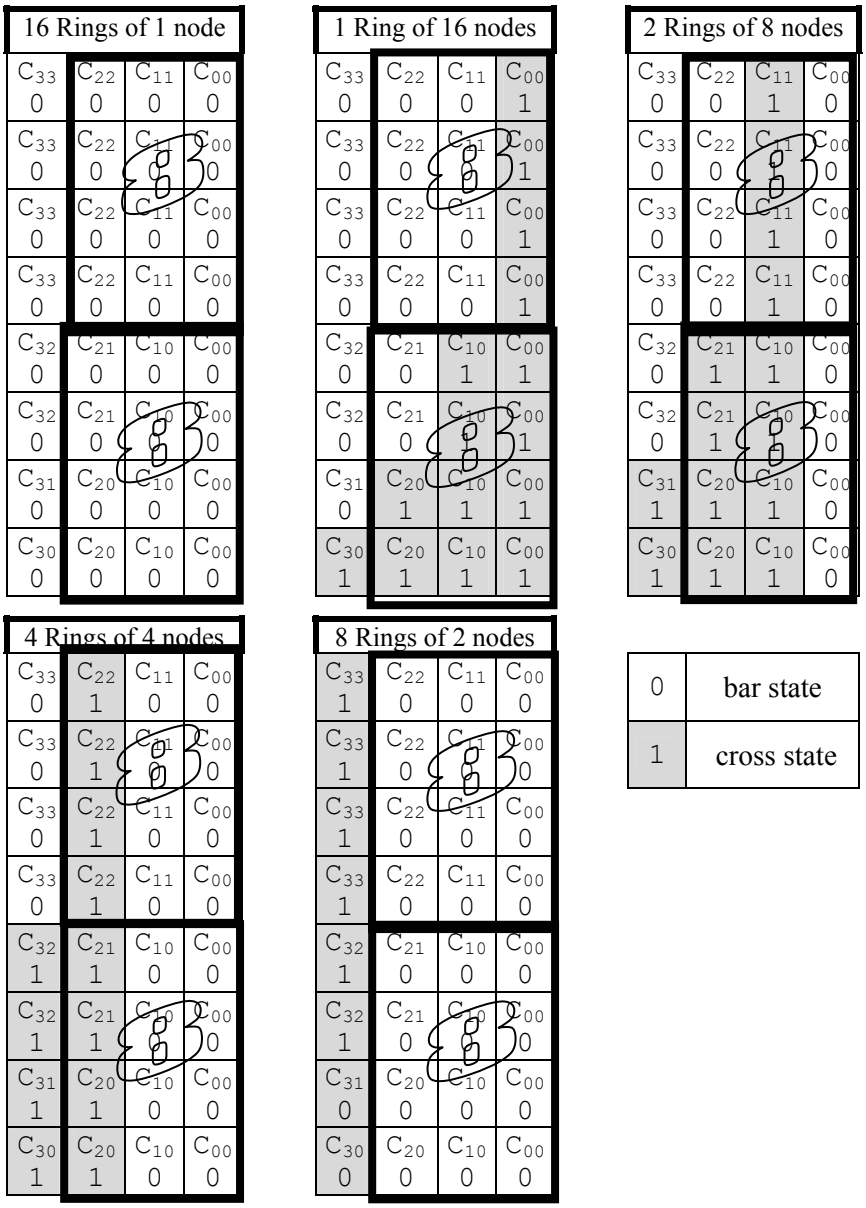


Figure 3.22: Controls for 16-Omega for each MultiRing configuration.

As illustrated in Figure 3.23, the control word for a 2*N*-Omega contains the control word for an *N*-Omega. However, in practice, extensive rewiring of inter-stage connections and exchange elements' control bits are required.

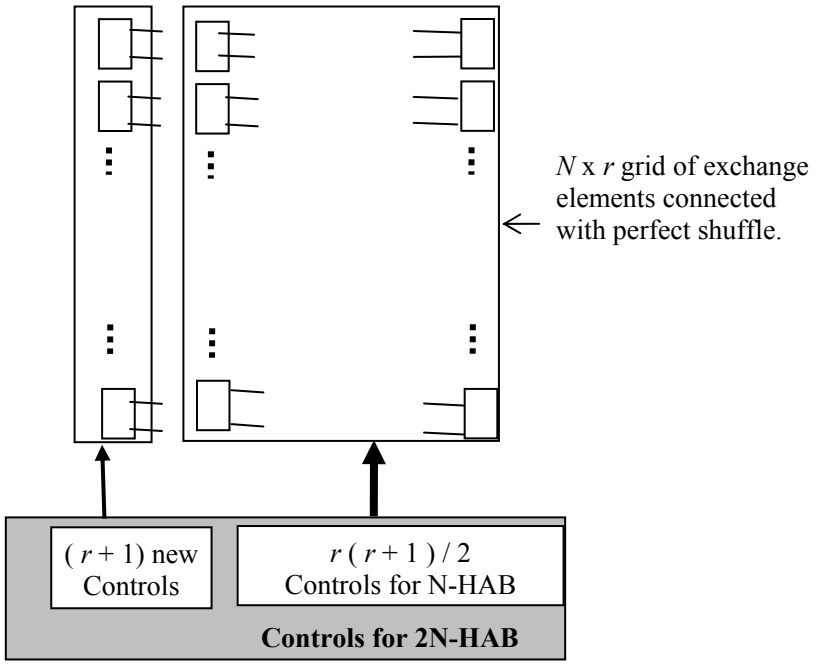


Figure 3.23: Controls for a 2N-Omega.

All of the rings previously examined were clockwise. If counter-clockwise rings were required, the exchange elements would be set differently for 2 rings of 4 nodes (Figure 3.24) and 1 ring of 8 nodes (Figure 3.25). As with the HAB network, controlling and Omega network becomes more complex to allow both clockwise and counter-clockwise rings. An additional control bit is needed to allow both clockwise and counter-clockwise configurations.

In Figure 3.24 and Figure 3.25, the shaded exchange elements are set differently than for clockwise communication. In Figure 3.24, the paths in the counter clockwise ring of 4 nodes

from *A* to *G* and from *C* to *A* are highlighted. In Figure 3.25, the counter-clockwise paths from *A* to *H* and from *B* to *A* are highlighted.

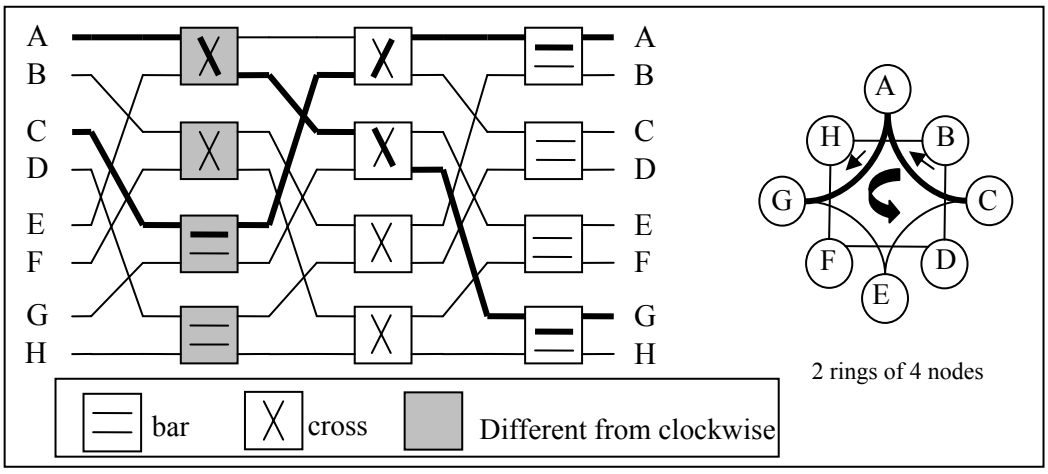


Figure 3.24: Two counter-clockwise rings of 4 on 8-Omega network.

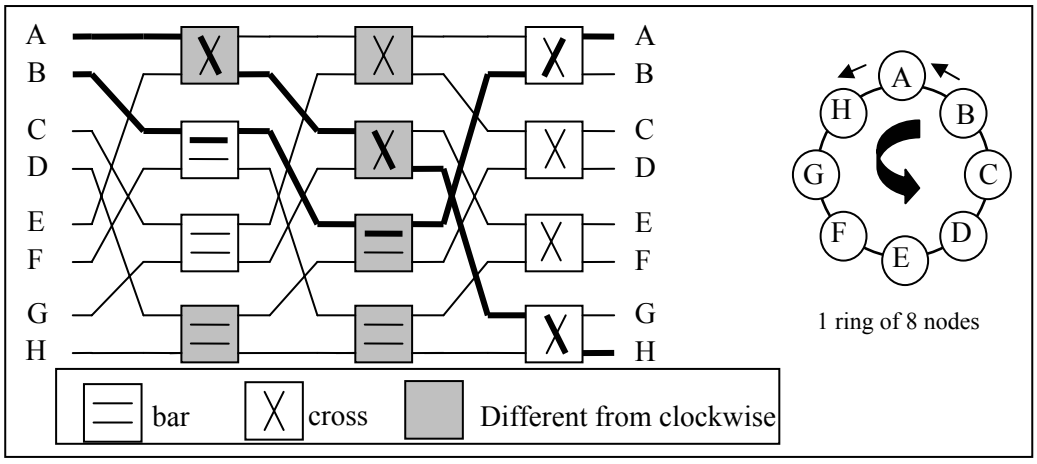


Figure 3.25: Counter-clockwise ring of 8 nodes on 8-Omega network.

3.3 Concluding Remarks

This chapter has presented issues for mapping the MultiRing onto the Butterfly and Omega networks. Conflicting states on the Butterfly network resulted in an additional

permutation after the last stage, so the HAB network [30] was used. Even though mapping on the Omega network did not require additional permutations, extensive rewiring was required when increasing the number of processors. Also, both HAB and Omega networks used a synchronized control strategy that is not easily scalable when modifying the network to allow both clockwise and counter-clockwise rings and expanding the network to include more nodes. In order for the MultiRing to scale modularly, the switching control needs to easily adapt for expansion and bidirectional data flow. The next chapter focuses on constructing a multi-stage switching network with a simple controlling strategy designed specifically for the MultiRing.

CHAPTER 4 : MULTIRING SWITCH

While the MultiRing maps onto the Omega and the HAB networks, expanding the Omega network requires extensive rewiring, and both networks have complex control strategies for managing the network. In both *N*-Omega and *N*-HAB networks, where $N=2^r$, clockwise configurations require $r(r+1)/2$ controls, multiple controls may be used in each column of switching elements. Additional controls are needed to allow for both clockwise and counter-clockwise ring configurations. In this section, two interconnection schemes with a simple control strategy are presented, ASB [12] and AWE MultiRing switches.

Both ASB and AWE MultiRing switches require only *r* controls to manage the network. Only one control is needed for each column. These networks use a control word based on the number of *nodes* on a ring in a configuration. In contrast, the Omega and HAB networks control word is based on the number of *rings* in a configuration. In a given MultiRing configuration, *R* rings contain *D* nodes. *D* can be expressed in binary in $r+1$ bits, $D = D_rD_{r-1}...D_1D_0$. For example in a network with 2^3 nodes, there can be 1, 2, 4 or 8 nodes on a ring – 0001, 0010, 0100 or 1000 in binary respectively. The notation C_i represents a single control bit, where *i* is a column number from *0* to *r-1*. For both designs presented, the control for column *i* is defined as $C_i = D_{i+1}$.

The ASB and AWE switches differ regarding the type of switching elements involved, the ordering of the columns in the multi-state network and the interconnection between columns of switching elements. Details of Arabnia and Smith's ASB switch [12] are given in section 4.1. Details of a newly proposed AWE switch are provided in section 4.2. In section 4.3, the ASB

and AWE switches are compared, and the benefits of both are given. In section 4.4, details to construct clockwise and counter clockwise unidirectional switches and fully bi-directional switches are provided. Finally, section 4.5 contains a brief description of the reconfiguration strategies for resetting the switch controls.

4.1 ASB Network and MultiRing Switch

Arabnia and Smith [12] designed a multistage interconnection network specifically for the MultiRing in which the basic building block is a 3x2 switching element which accepts three inputs and produces two outputs. The switching element is composed of two multiplexers controlled by the same control signal. One input is shared by both multiplexers as depicted in Figure 4.1.

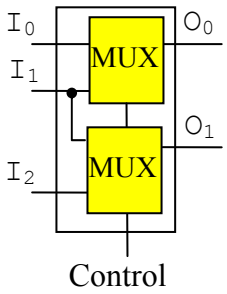


Figure 4.1: 3x2 switching element

The switching element can be set to two states, shift and bar. As illustrated in Figure 4.2, if the bar state is selected, the middle and bottom inputs, A_1 and A_2 are passed to the output. If the switching element is set to the shift state, the top and middle inputs, A_0 and A_1 are passed to the output.

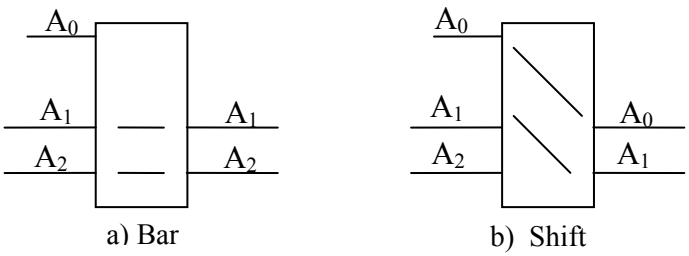


Figure 4.2 States of 3x2 switching element

An integral part of the design is a collection of switching elements where the inputs are "chained" together to form a ring. The bottom input of one switching element is also used as the top input of the element below it. To close the ring, the bottom input of the last switching element is connected to the top input for the first switch. A chain of *N* switches will be referred to as an *N*-chain.

A single signal controls all of the switching elements in an *N*-chain. A 4-chain is depicted in Figure 4.3. If all of the elements are set to bar state, the inputs are simply passed to the output. However, if all of the switches are set to shift state, the inputs are shifted down one position and the bottom input rotates to the top.

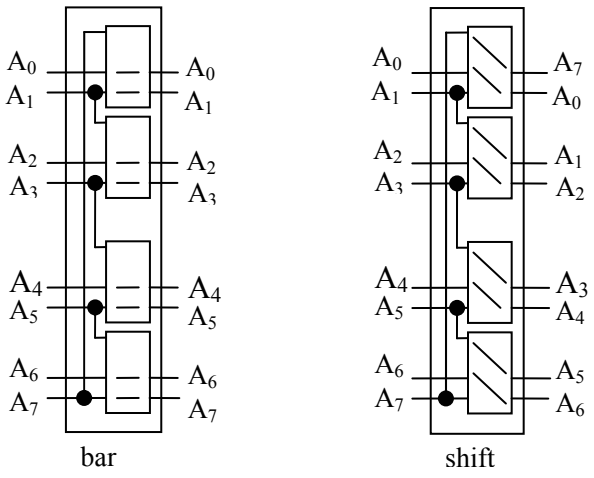


Figure 4.3: 4-chain of switching elements

The collection of 3x2 switching elements is arranged in a modified butterfly network with $N/2$ rows and r columns of switching elements. This network will be referred to as an ASB (**A**rabnia and **S**mith **B**utterfly) network. An N-ASB network accepts $N=2^r$ inputs and produces N outputs. It can be recursively constructed with two $N/2$ ASB networks and an $N/2$-chain of additional switching elements. The output of the $N/2$-chain connects to the inputs of the $N/2$ ASB networks using an inverse perfect shuffle (See Figure 4.4).

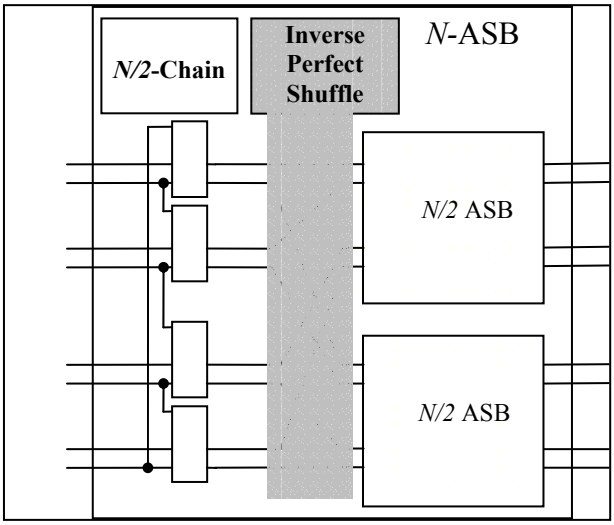


Figure 4.4: Recursive structure of ASB network

A 4-ASB constructed from two 2-ASB networks and an 8-ASB constructed from two 4-ASB network are provided in Figure 4.5 a and b. The butterfly interconnection pattern is most apparent if a chain of switching elements is viewed as a single unit (See Figure 4.5c).

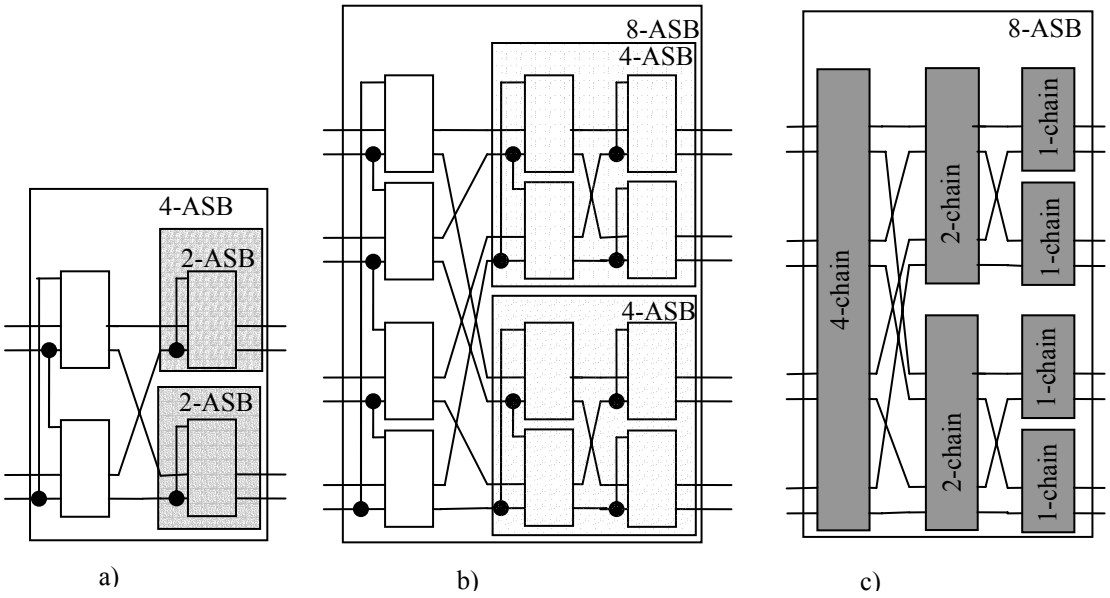


Figure 4.5: Constructing 4-ASB and 8-ASB using a butterfly interconnection pattern.

The ASB switch designed by Arabnia and Smith[12] uses an *N*-ASB network and shuffles the output according to the reverse of the nodes' binary id. Each node in an $N=2^r$ MultiRing has a node ID, *A* that can be expressed in binary in *r* bits where $A = a_{r-1}a_{r-2}a_{r-3}\ldots a_2a_1a_0$. Using a reverse shuffle, a node with id $a_{r-1}a_{r-2}a_{r-3}\ldots a_2a_1a_0$ is connected to the node with id $a_0a_1a_2\ldots a_{r-3}a_{r-2}a_{r-1}$. The reverse shuffle of 8 nodes numbered from 0 to 7 is given in Figure 4.6.

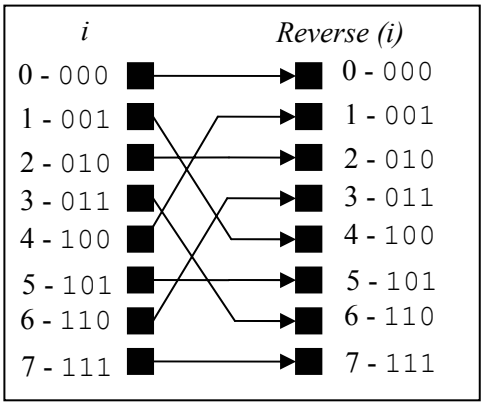
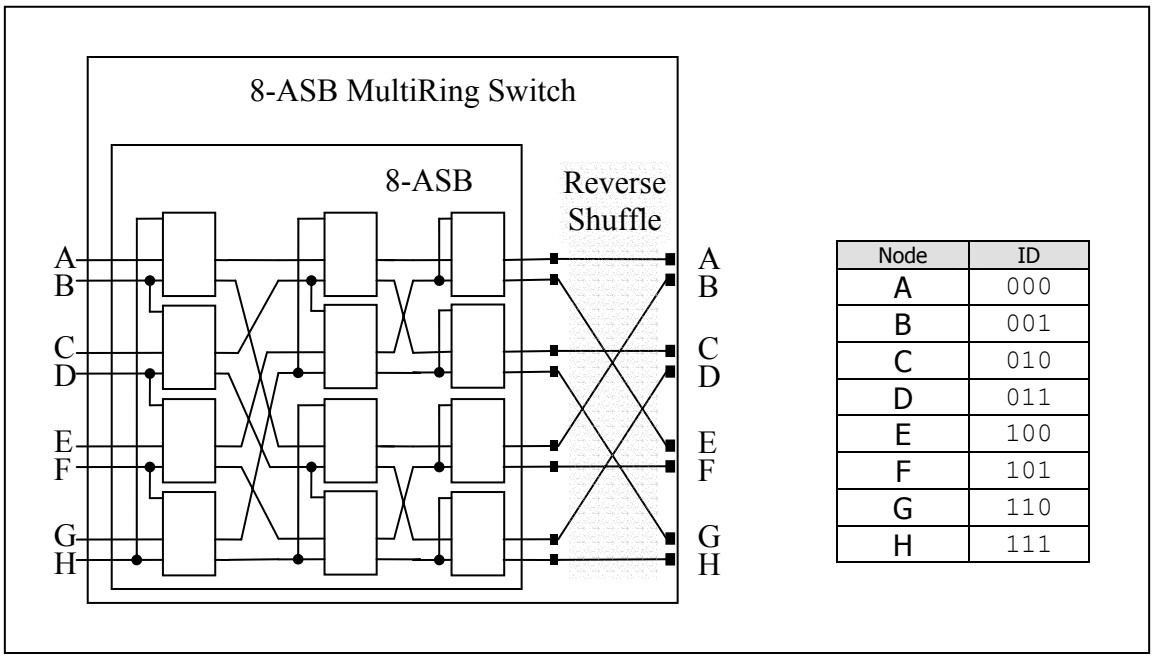


Figure 4.6: Reverse Shuffle

An 8-ASB switch containing an 8-ASB along with the interconnection for a reverse shuffle is provided in Figure 4.7. The nodes are labeled *A-H* with the ids 000-111.



Node	ID
A	000
B	001
C	010
D	011
E	100
F	101
G	110
H	111

Figure 4.7: 8-ASB MultiRing switch

In Figure 4.8- Figure 4.9 the four different configurations of the 8-node MultiRing are mapped onto an 8-ASB switch network. A MultiRing is said to "map onto a network" by setting the switching elements to establish the required communication for each configuration of the MultiRing. The 8-ASB switch can be configured for:

- 8 Rings of 1 node (Figure 4.8)
- 1 Ring of 8 nodes (Figure 4.9)
- 2 Rings of 4 nodes (Figure 4.10)
- 4 Rings of 2 nodes (Figure 4.11)

Two paths on each of the ring configurations are highlighted.

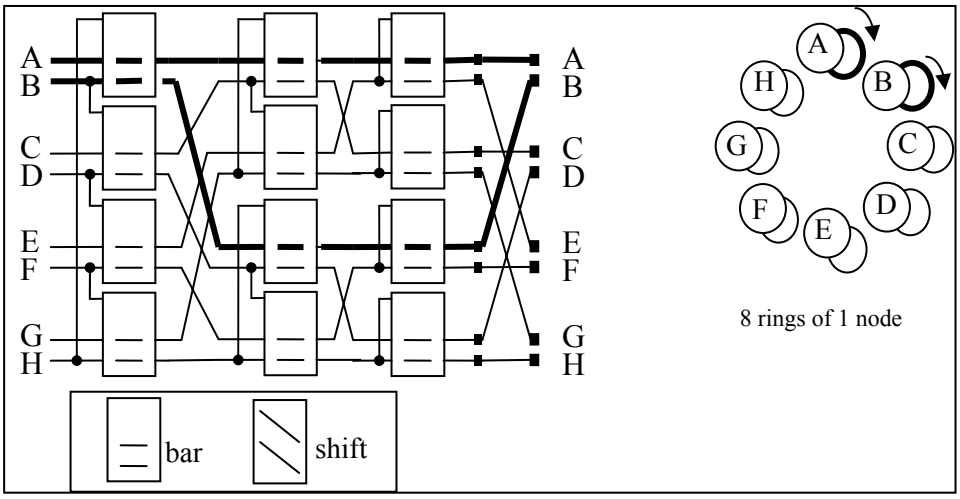


Figure 4.8: 8 rings of 1 node on 8-ASB switch

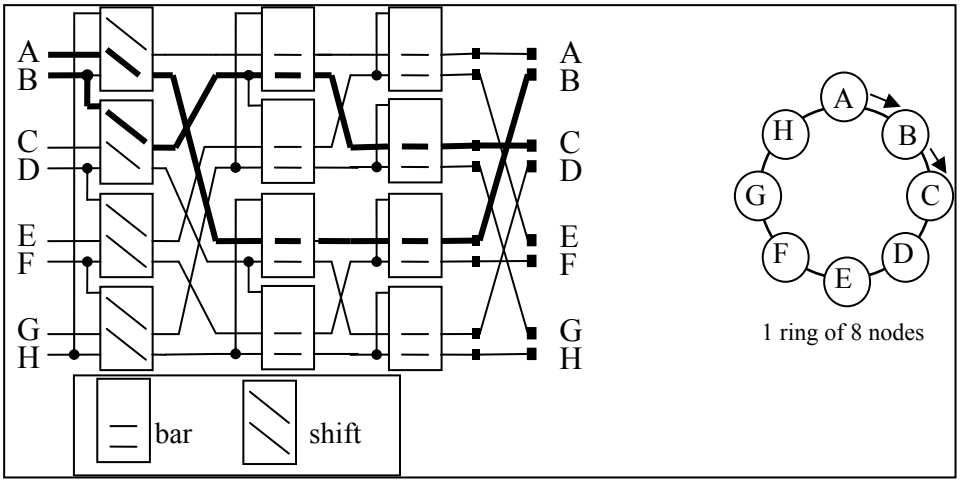


Figure 4.9: 1 ring of 8 nodes on 8-ASB switch

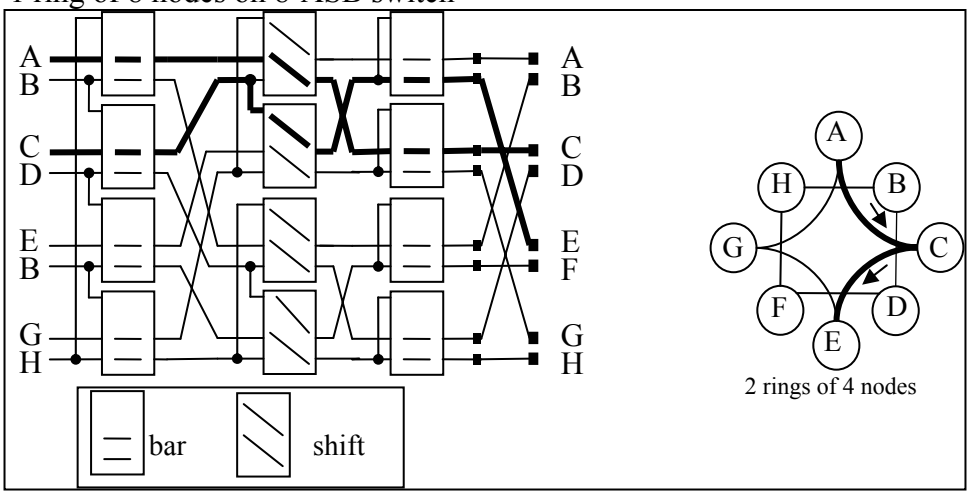


Figure 4.10: 2 rings of 4 nodes on 8-ASB switch

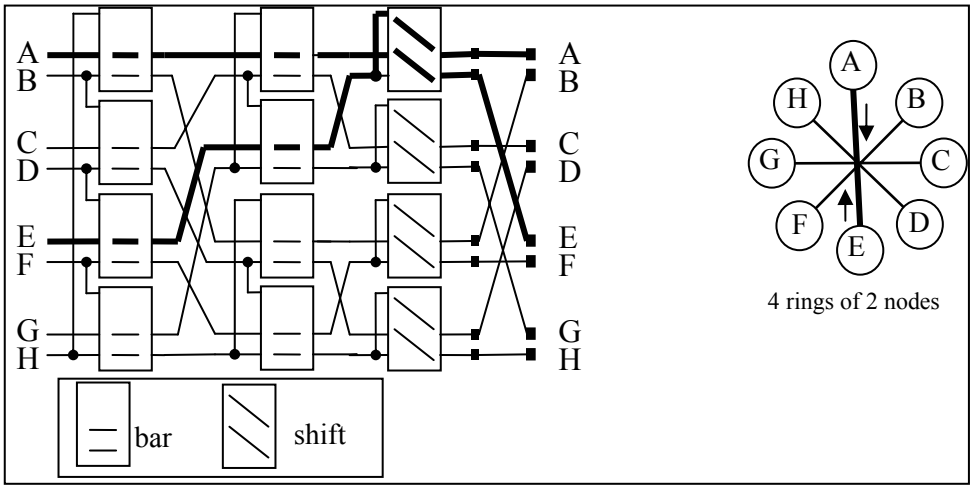


Figure 4.11: 4 rings of 2 nodes on 8-ASB switch

In the ASB switch design, the columns are numbered from *right* to *left*. As the network expands to include additional nodes, a new column of switching elements is added to the left of the existing network. Each column requires one bit of the control word. Column *i* requires control bit C_i (see Figure 4.12). If $C_i = 1$ all of the switching elements in the column are set to shift state; otherwise if $C_i = 0$ all of the switching elements are set to bar state.

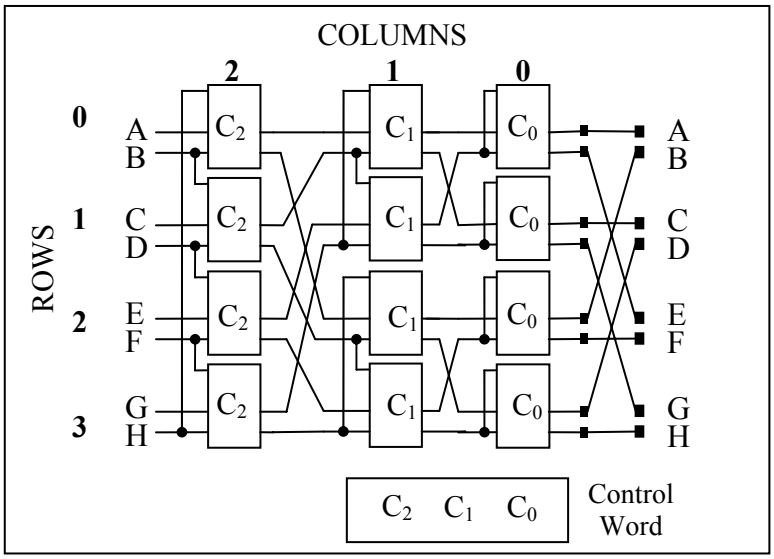


Figure 4.12: Location of controls for 8-ASB switch

The controls for an 8-ASB switch establishing the four configurations of an 8- node MultiRing are in Figure 4.13. Also, the value of each switching element's control bit is provided. The 8 rings of 1 node control settings correspond to Figure 4.8. The 1 ring of 8 nodes control settings correspond to Figure 4.9. The 2 rings of 4 nodes control settings correspond to Figure 4.10. The 4 rings of 2 nodes control settings correspond to Figure 4.11.

Number of Nodes	D	Controls for 8-ASB Switch
	$D_3\ D_2\ D_1\ D_0$	$C_2\ C_1\ C_0$
1	**0 0 0** 1	0 0 0
2	**0 0 1** 0	0 0 1
4	**0 1 0** 0	0 1 0
8	**1 0 0** 0	1 0 0

8 rings of 1 node		
C_2	C_1	C_0
0	0	0
0	0	0
0	0	0
0	0	0

1 ring of 8 nodes		
C_2	C_1	C_0
1	0	0
1	0	0
1	0	0
1	0	0

2 rings of 4 nodes		
C_2	C_1	C_0
0	1	0
0	1	0
0	1	0
0	1	0

4 rings of 2 nodes		
C_2	C_1	C_0
0	0	1
0	0	1
0	0	1
0	0	1

0	bar state

1	shift state

Figure 4.13: Controls for 8-ASB switch for each MultiRing configuration

In general, when constructing a 2*N*-MultriRing Switch with two existing *N*-ASB networks, the existing control signals remain connected to the *N*-ASB networks. A new N-chain is added to the left of the existing networks and a new control signal C_r is added to manage it (See Figure 4.14).

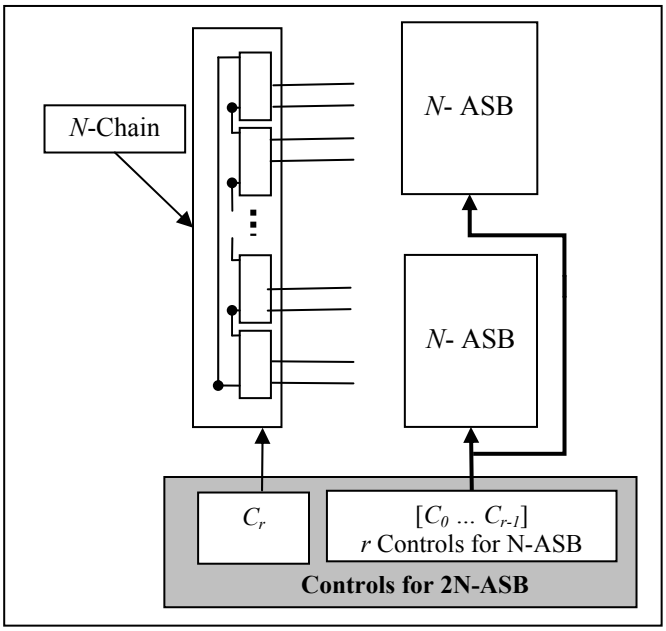


Figure 4.14: Controls for 2N-ASB switch

The controls for a 16-ASB switch for each configuration in a 16-MultiRing are provided in Figure 4.15. The locations of the 8-ASB networks are identified with a dark square and the number 8 in the center. Notice that control signals in the 8-ASB switching elements are contained in the 16-MultiRing. For example, the setting for the 8-ASB switching elements for the configuration with 2 nodes on the 16-ASB are the same as 2 nodes on the 8-ASB.

Number of Nodes	D	Controls for 16-MultiRing Switch
	$D_4 D_3\ D_2\ D_1 D_0$	$C_3\ C_2\ C_1\ C_0$
1	**0 0 0 0** 1	0 0 0 0
2	**0 0 0 1** 0	0 0 0 1
4	**0 0 1 0** 0	0 0 1 0
8	**0 1 0 0** 0	0 1 0 0
16	**1 0 0 0** 0	1 0 0 0

16 Rings of 1 node

C_3	C_2	C_1	C_0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

1 Ring of 16 nodes

C_3	C_2	C_1	C_0
1	0	0	0
1	0	0	0
1	0	0	0
1	0	0	0
1	0	0	0
1	0	0	0
1	0	0	0
1	0	0	0

2 Rings of 8 nodes

C_3	C_2	C_1	C_0
0	1	0	0
0	1	0	0
0	1	0	0
0	1	0	0
0	1	0	0
0	1	0	0
0	1	0	0
0	1	0	0

4 Rings of 4 nodes

C_3	C_2	C_1	C_0
0	0	1	0
0	0	1	0
0	0	1	0
0	0	1	0
0	0	1	0
0	0	1	0
0	0	1	0
0	0	1	0

8 Rings of 2 nodes

C_3	C_2	C_1	C_0
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1

0	bar state
1	shift state

Figure 4.15: Controls for 16-ASB switch for each MultiRing configuration

A 16- ASB switch is given in Figure 4.16. The nodes are labeled from P_0 to P_{15} and the output of the 16-ASB is labeled D_0 to D_{15}. After a reverse shuffle, the paths from P_i to P_i for all $0 \leq i < 16$ are created when all elements in the ASB switch are set to the bar state.

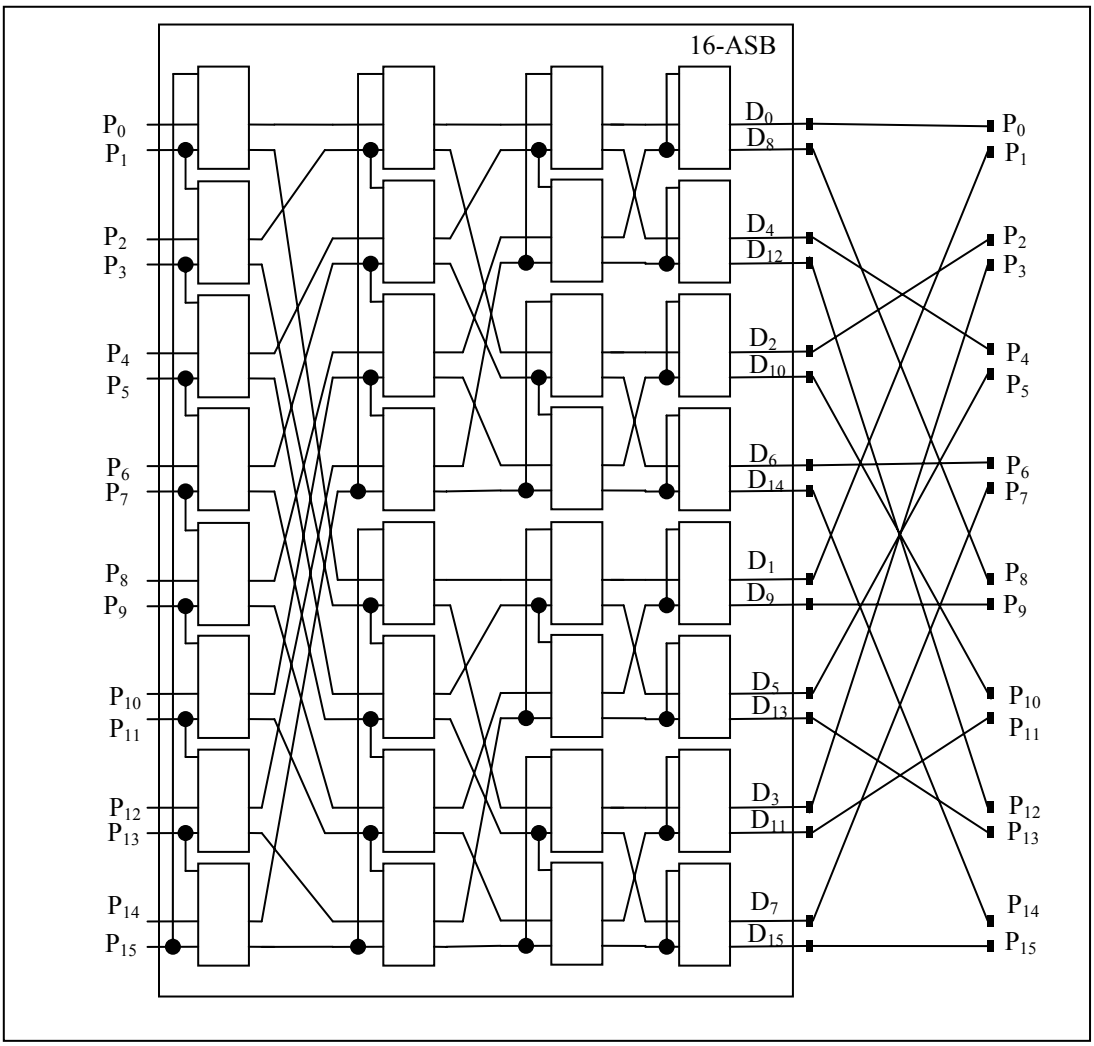


Figure 4.16: 16-ASB switch

4.2 AWE Switch

The design of the AWE Switch is centered on maintaining the *same* rings in a MultiRing with *N* nodes as one with *2N* nodes. The original design for the MultiRing switch focused on matching the order nodes are connect to the switch to the order nodes are connected in a single

ring of *N* nodes. For example, if the nodes *A*, *B*, *C, D* were connected to a 4-ASB MultiRing switch or the 4-HAB as in Figure 4.17, the ring of four nodes would be in the order *A*-*B*-*C*-*D* (Figure 4.17 b). This is an unnecessary restriction. With the AWE Switch the ring of four nodes would be *A*-*C*-*B*-*D*. (Figure 4.17c) The design of the AWE Switch focuses on connecting pairs of nodes and combining pairs to create larger rings. The rings *A*-*B* and *C*-*D* are first created and then joined together to create *A*-*C*-*B*-*D*.

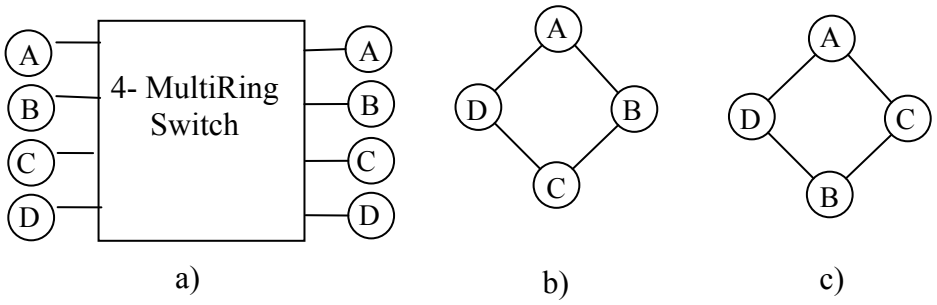


Figure 4.17: Sequence on a ring of 4 nodes

The basic component of the AWE switch is a 4x2 switching element that accepts 4 inputs and produces two outputs. As illustrated in Figure 4.18, the switching element is composed of two multiplexers that are controlled by the same control signal. Each multiplexer accepts a pair of inputs and produces a single output.

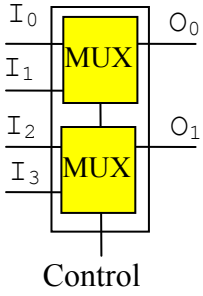


Figure 4.18: 4x2 switching element

The two states of the switching element are called bar and shift; however, these are different than for the 3x2 switching element in the ASB switch. When the control signal is set to 0, the switching element is in the bar state where the top input from both multiplexers passes to the output. When the control signal is set to 1, the switching element is in the shift state. In the shift state, the bottom input from both multiplexers passes to the output (see Figure 4.19).

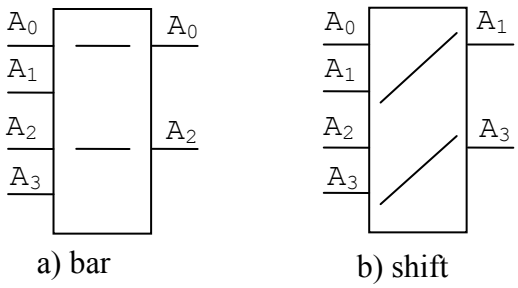


Figure 4.19: States of a 4x2 switching element

An *N*-AWE Switch can be recursively built from two *N*/2 AWE switches and a new collection of *N/2* switching elements. The additional switching elements are added to the *right*

of the existing network. Therefore, the output of the *N*/2 AWE switches connects to the input of the new column of switching elements. As illustrated in Figure 4.20, there are twice as many inputs as outputs. Each output must connect to two inputs. The outputs are numbered from 0 to *N*-1 and the inputs are numbered from 0 to 2*N*-1.

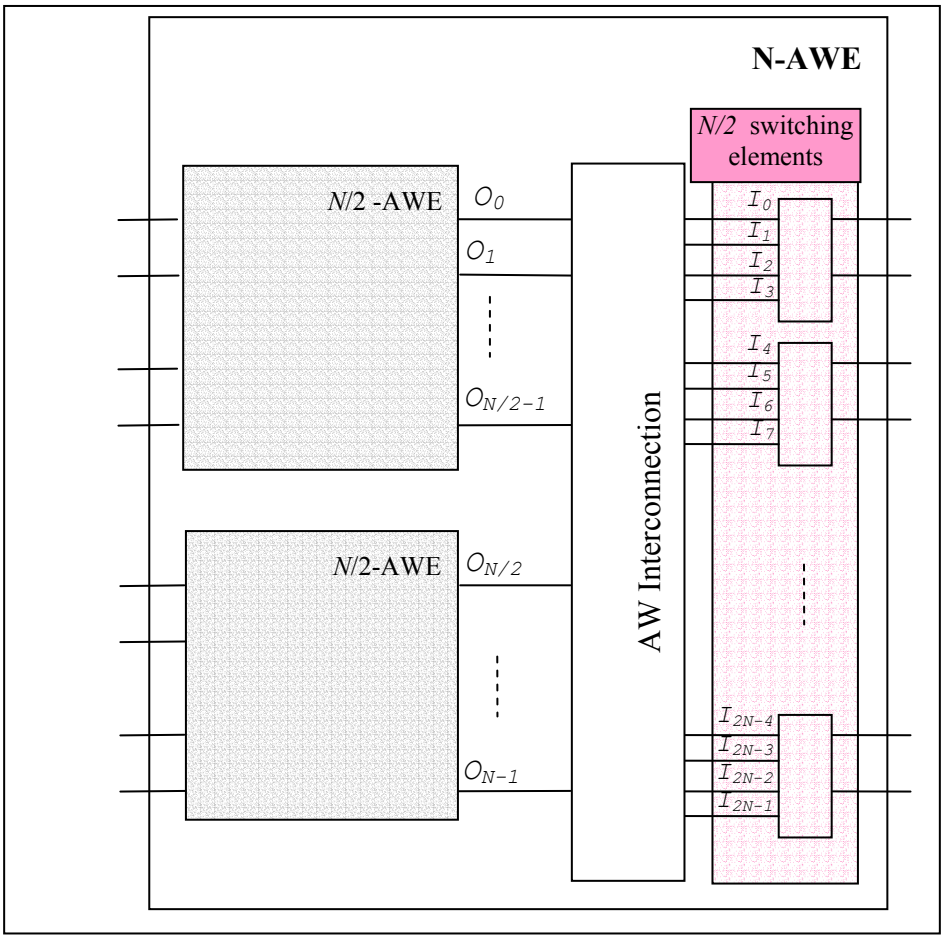


Figure 4.20: Recursive structure of AWE Switch

The AWE-interconnection scheme connects output, O_i to inputs $I_{f(i)}$ and $I_{g(i)}$ where $0 \leq i < N$ and *f(i)* and *g(i)* are defined as follows:

$$f(i) = 2i$$

$$g(i) = \begin{cases} 1 & \text{for } i = N\text{-}1 \\ \frac{N}{b} + 2\left(i - \frac{(b-1)N}{b}\right) + 1 & \text{for } 0 \le i < N\text{-}1 \end{cases}$$

where $b = 2^a$ and a represents the number of preceding ones in the binary representation of i. (i.e., starting at bit r-1, a is the number of 1's in binary (i) until a 0 bit is encountered.) For example in a network with 2^3 nodes, $a = 0$ for binary (3), `011`, $a = 1$ for binary (4), `100`, and $a =$ 2 for binary (6), `110`. However, in a network of 2^4 nodes, $a = 0$ for binary (3) ,`0011`, binary (4), `0100`, and binary (6), `0110`.

As illustrated in Figure 4.21, *f(i)* connects outputs to the *top* of input pairs(the even numbered inputs) and *g(i)* connects outputs to *bottom* of input pairs (the odd numbered inputs). The interconnection defined by *g(i)* separates the outputs into r+1 groups and connects each group of outputs in consecutive order to a corresponding group of inputs. In the first group, there are 2^{r-1} outputs, O_0 to $O_{N/2-1}$. In the next group there are 2^{r-2} outputs, $O_{N/2}$ to $O_{3N/4-1}$. The pattern continues on until there are two groups of 1 with one output each, O_{N-2} and O_{N-1}. In total there are 2^r outputs; $2^r = 2^{r-1} + 2^{r-2} + \ldots + 2^0 + 1 = 1 + \sum_{x=0}^{r-1} 2^x$.

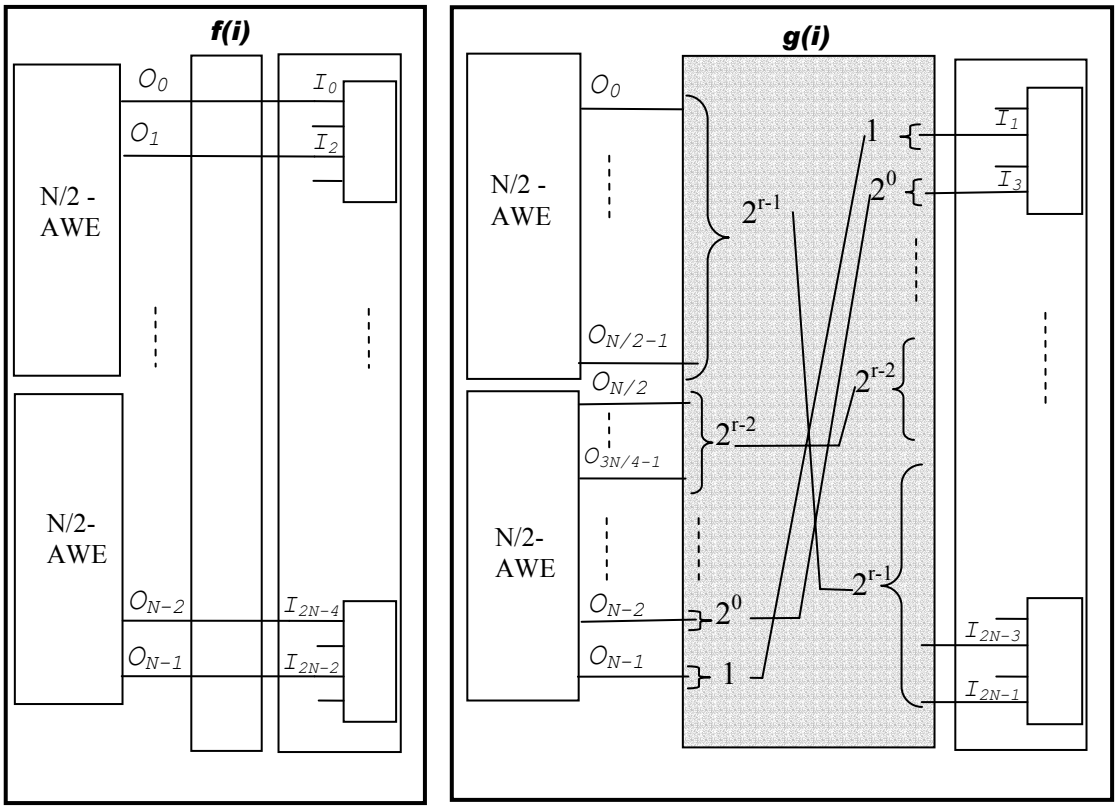


Figure 4.21: AWE Interconnection Scheme

In Figure 4.22 a 4-AWE Switch is constructed from two 2-AWE Switches. Both functions in the interconnection scheme are evaluated and the corresponding links appear on the 4-AWE Switch. Additional examples of the AWE Interconnection scheme are given in Figure 4.23 where an 8-AWE switch is constructed from two 4-AWE switches and in Figure 4.24 where a 16-AWE switch is constructed from two 8-AWE switches.

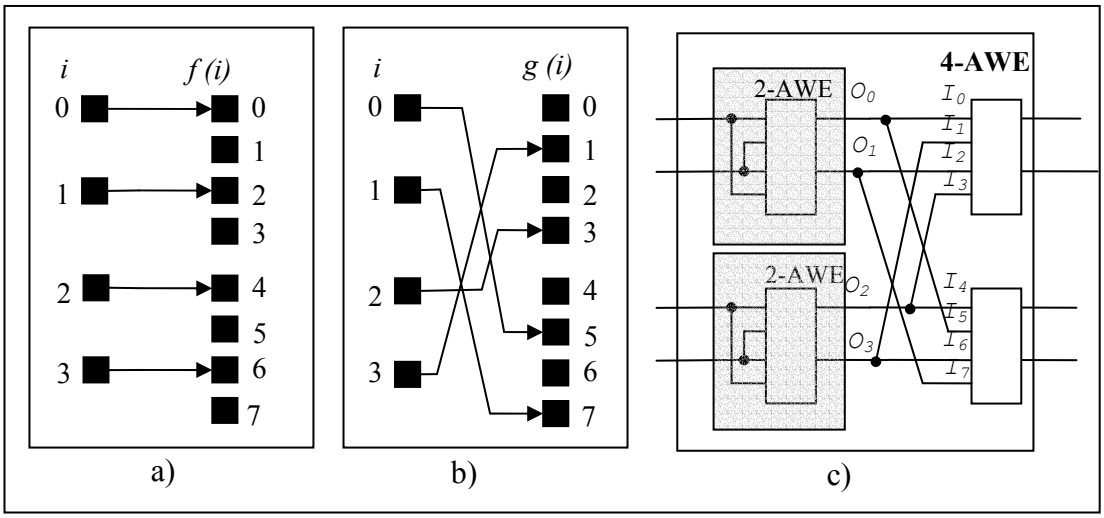


Figure 4.22: Constructing 4-AWE Switch with two 2-AWE Switches (N=4)

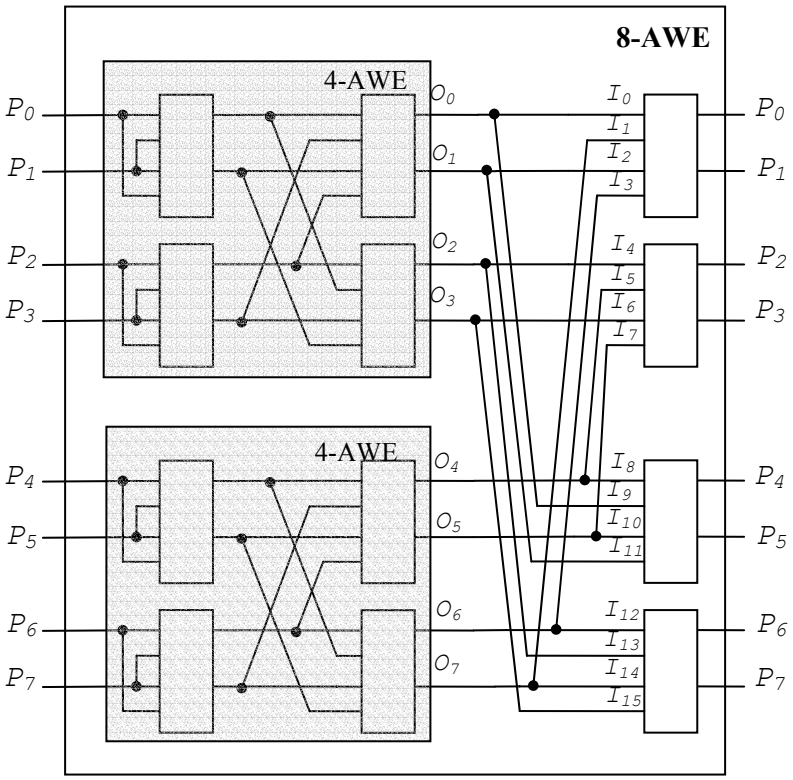


Figure 4.23: Constructing an 8-AWE Switch with two 4-AWE Switches (N=8)

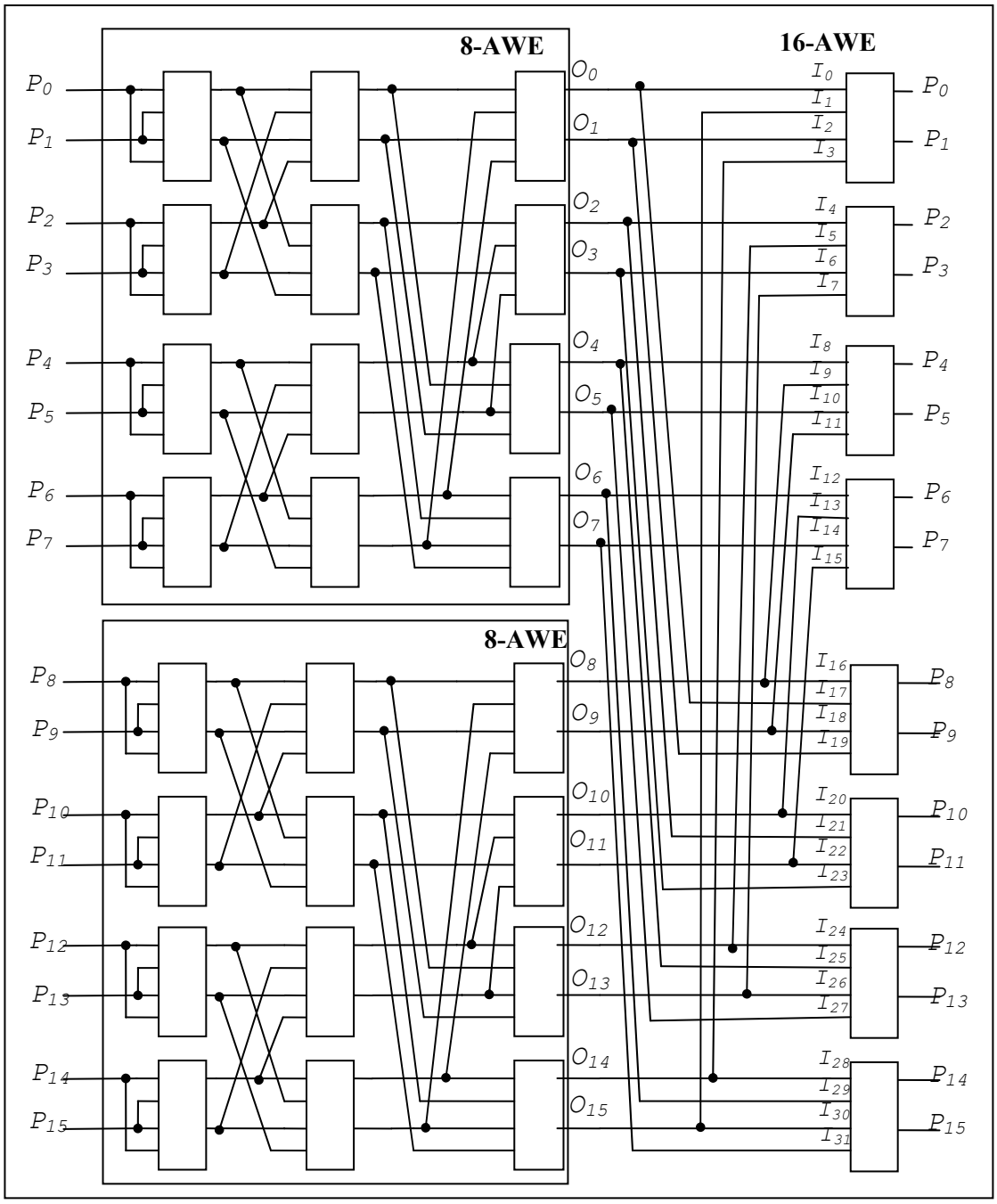


Figure 4.24: Constructing a 16-AWE switch with two 8-AWE switches (N=16)

An 8-node MultiRing with nodes labeled *A*-*H*, connected to an AWE switch as in Figure 4.25a has the four configurations in Figure 4.25b.

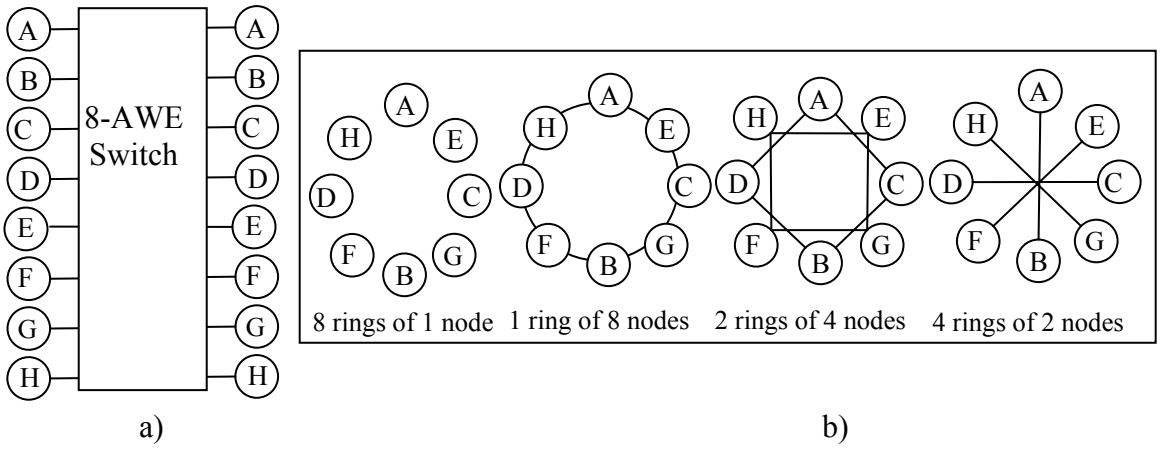


Figure 4.25: MultiRing configurations using the AWE switch.

In Figure 4.26- Figure 4.29 the four different configurations of the 8-node MultiRing are mapped onto an 8-AWE Switch network. The 8-AWE Switch can be configured for:

- 8 Rings of 1 node (Figure 4.26)
- 1 Ring of 8 nodes (Figure 4.27)
- 2 Rings of 4 nodes (Figure 4.28)
- 4 Rings of 2 nodes (Figure 4.29)

Two paths on each of the ring configurations are highlighted.

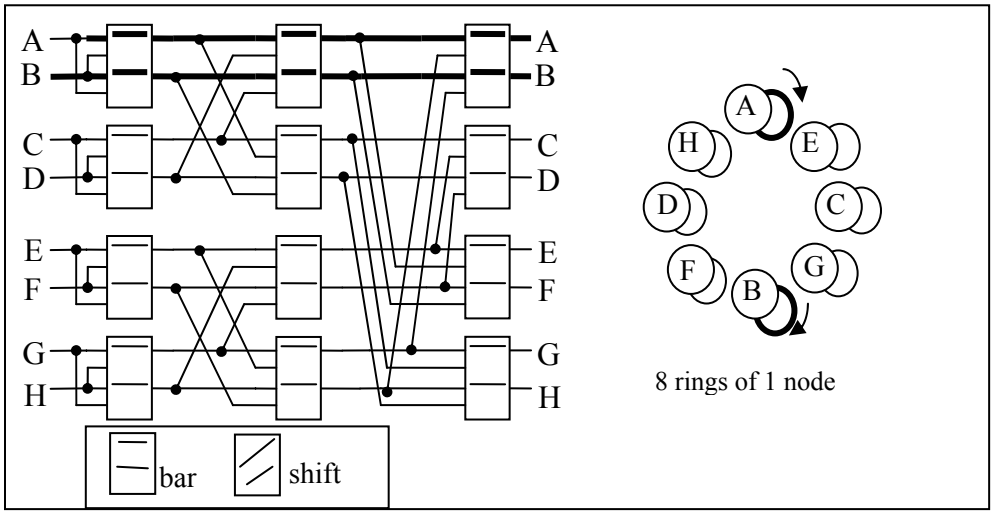


Figure 4.26 8 rings of 1 node on 8-AWE Switch.

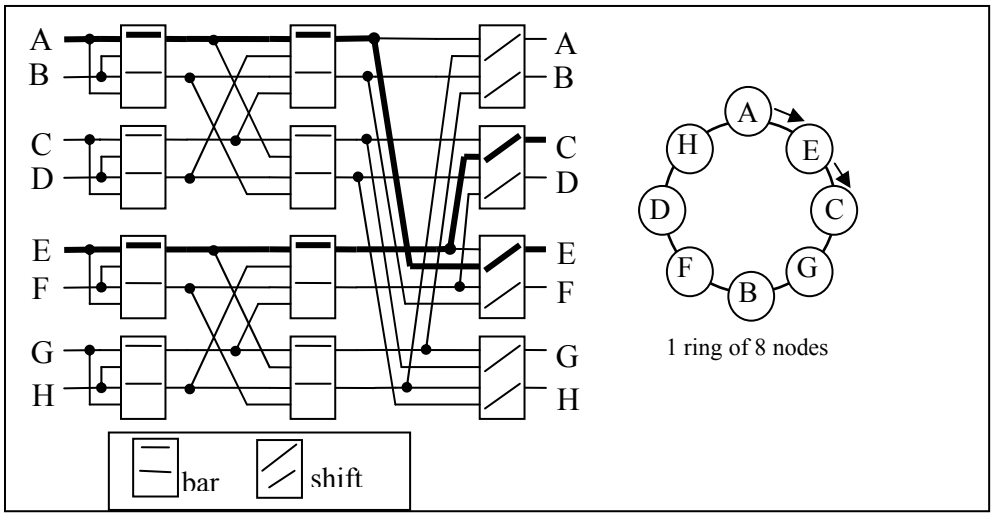


Figure 4.27: 1 ring of 8 nodes on 8-AWE Switch

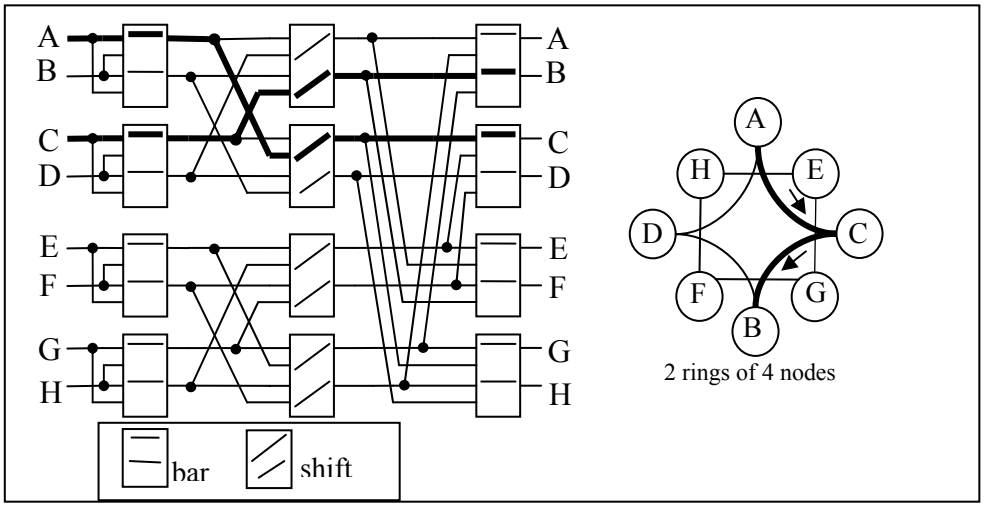


Figure 4.28: 2 Rings of 4 nodes on 8-AWE Switch.

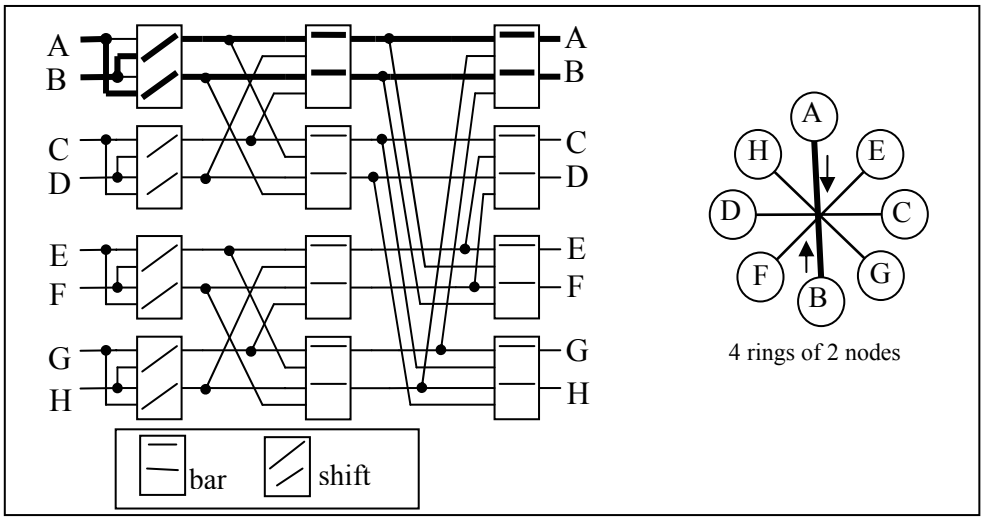


Figure 4.29: 4 rings of 2 nodes on 8-AWE Switch

In contrast to the ASB switch, the columns in the AWE switch are labeled from *left* to *right.* Each column has one control bit, column *i* has control bit C_i (see Figure 4.30). If $C_i = 1$ all of the switching elements in the column are set to shift state; otherwise if $C_i = 0$ all of the switching elements are set to bar state.

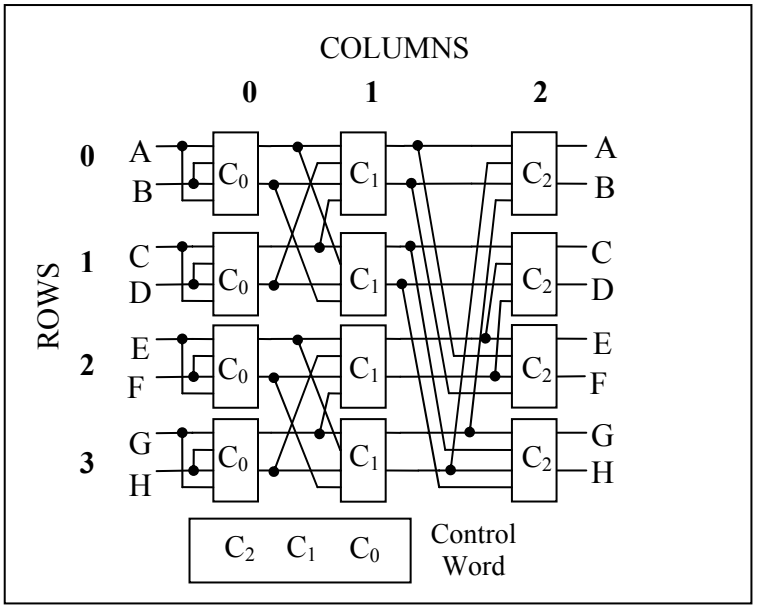


Figure 4.30: Location of controls in 8-AWE Switch

The controls for an 8-AWE Switch establishing the four configurations of an 8- node MultiRing are in Figure 4.31. Also, the value of all switching elements' control bit is provided. The 8 rings of 1 node control settings correspond to Figure 4.26. The 1 ring of 8 nodes control settings correspond to Figure 4.27. The 2 rings of 4 nodes control settings correspond to Figure 4.28. The 4 rings of 2 nodes settings correspond to Figure 4.29.

Number of Rings	Controls for 8-AWE Switch
	C_2 C_1 C_0
0	0 0 0
1	0 0 1
2	0 1 0
4	1 0 0

8 rings of 1 node		
C_0 0	C_1 0	C_2 0
C_0 0	C_1 0	C_2 0
C_0 0	C_1 0	C_2 0
C_0 0	C_1 0	C_2 0

1 Ring of 8 nodes		
C_0 1	C_1 0	C_2 0
C_0 1	C_1 0	C_2 0
C_0 1	C_1 0	C_2 0
C_0 1	C_1 0	C_2 0

2 Rings of 4 nodes		
C_0 0	C_1 1	C_2 0
C_0 0	C_1 1	C_2 0
C_0 0	C_1 1	C_2 0
C_0 0	C_1 1	C_2 0

4 Rings of 2 nodes		
C_0 0	C_1 0	C_2 1
C_0 0	C_1 0	C_2 1
C_0 0	C_1 0	C_2 1
C_0 0	C_1 0	C_2 1

0	bar state

1	shift state

Figure 4.31: Controls for 8-AWE Switch for each MultiRing configuration

In general, when constructing a 2*N*-AWE Switch with two existing *N*-AWE networks, the existing control signals remain connected to the *N*-AWE networks and a new control signal is added to manage the new column of switches (See Figure 4.14).

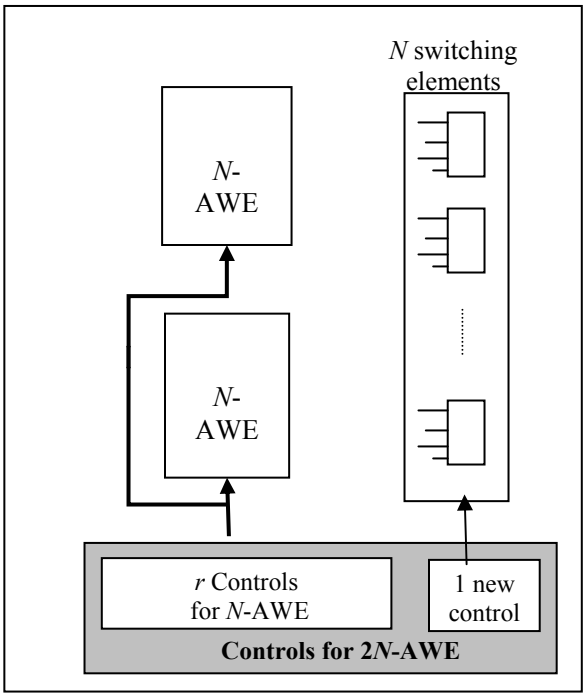


Figure 4.32: Controls for 2*N*-AWE Switch

The controls for a 16-AWE Switch for each configuration in a 16-MultiRing are provided in Figure 4.15. The locations of the 8-AWE networks are identified with a dark square and the number 8 in the center. As with the ASB switch, the control settings for an *N*-network are contained in a 2*N* network. Notice the settings for the 8-AWE switching elements for the configuration with 2 nodes on the 16-AWE Switch are the same as 2 nodes on the 8-AWE Switch.

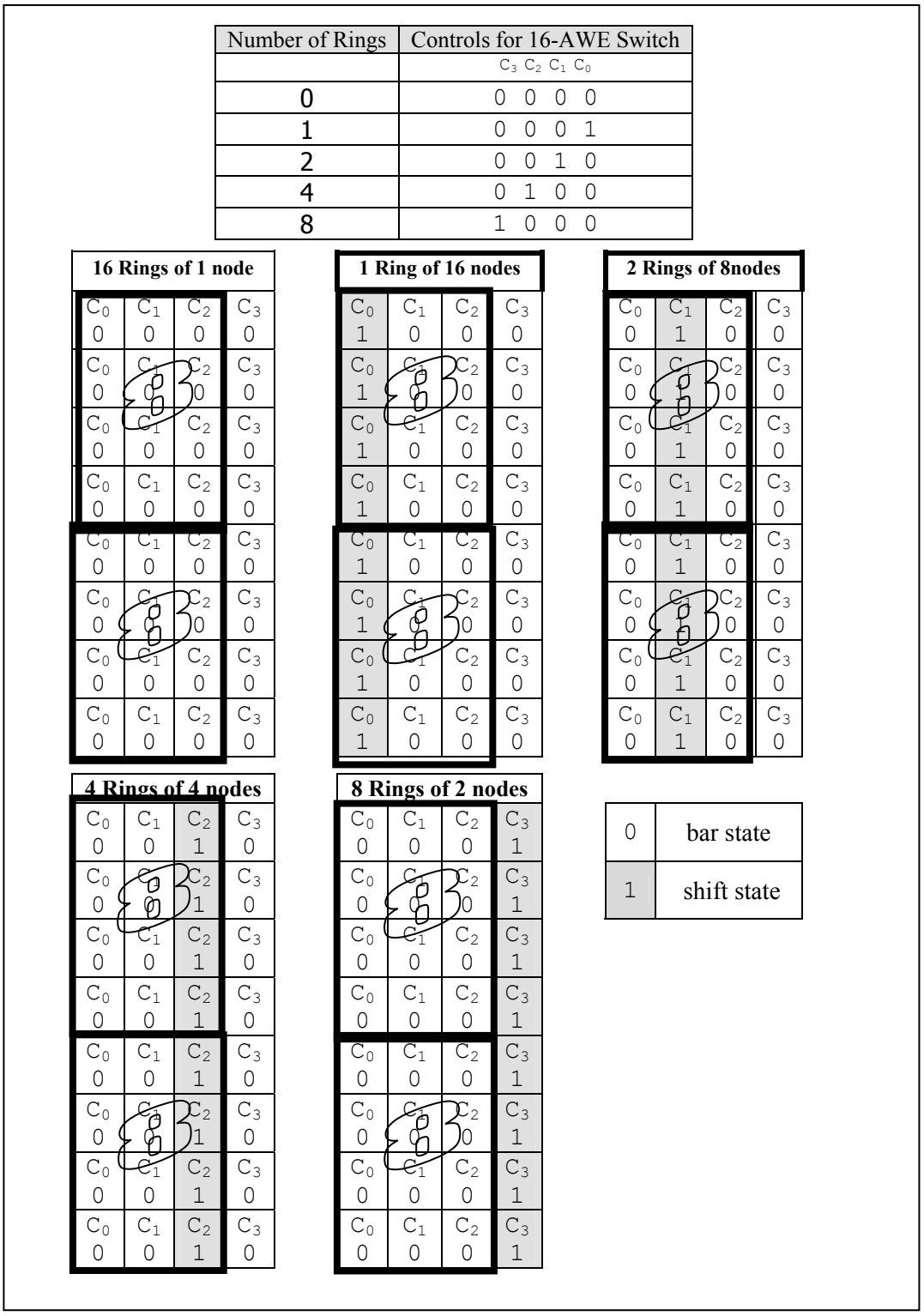


Figure 4.33: Controls for 16-AWE Switch for each MultiRing configuration

A 16- AWE switch is given in Figure 4.16.

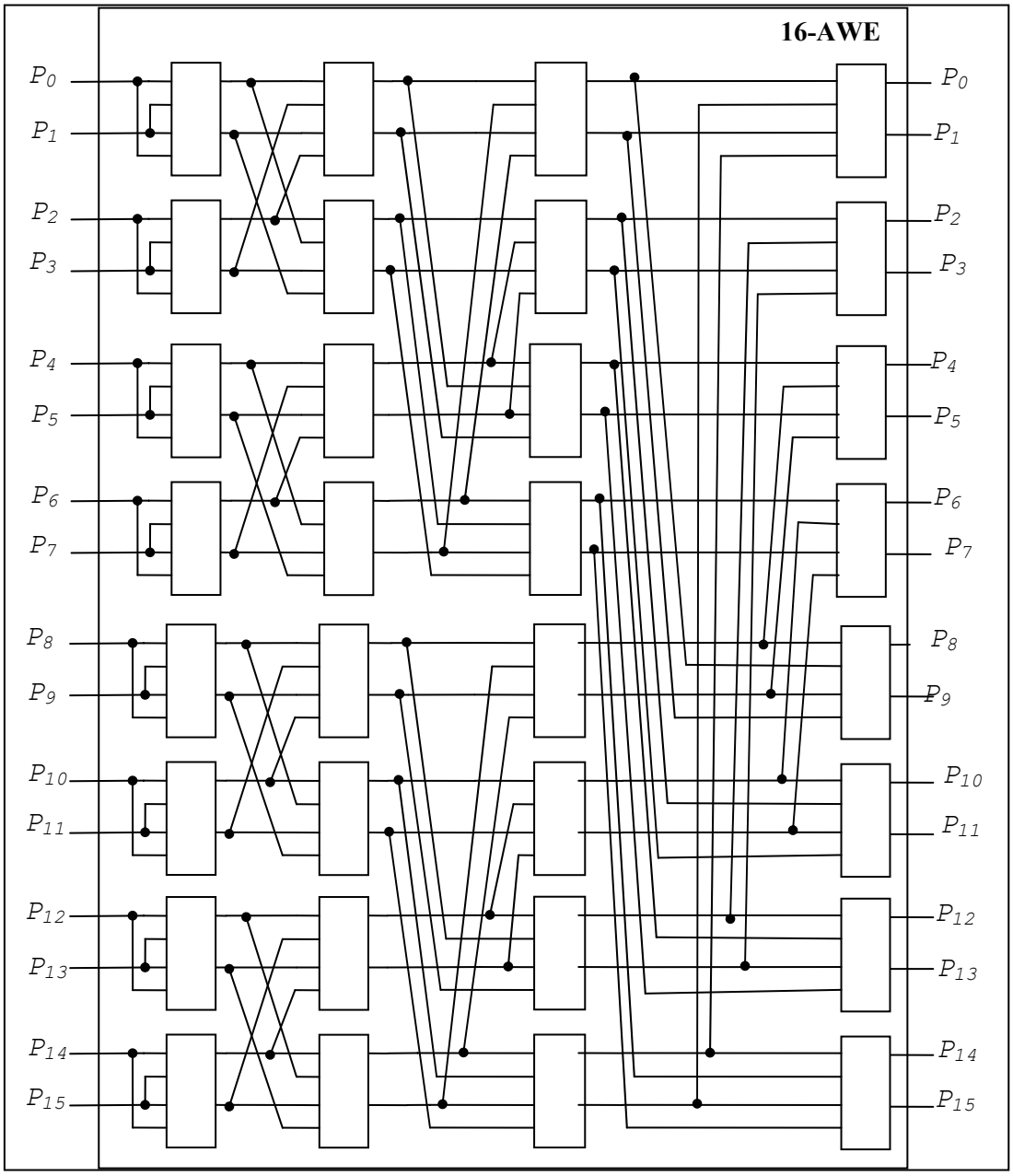


Figure 4.34: 16-AWE Switch

4.3 Comparing ASB and AWE Switches

The ASB and AWE switches have different types of switching elements; yet they have the same number of switching elements and the same number of logical gates. *(N log N)/2*

different switching elements are needed to map an *N*-node MultiRing, and both 3x2 and 4x2 switching elements are built with two multiplexers. However, the number of additional links required to construct an *N*-AWE switch with two *N*/2 switches is more than the links required to create an *N*-ASB switch.

The ASB switch, made with 3x2 switching elements, requires $\frac{3N}{2}$ additional links:

- $\frac{N}{2}$ additional links connect the bottom input of each switch to the top input of another to create the *N*/2-chain.
- *N* additional links connect outputs of *N*/2-chain to inputs of the two existing *N*/2 ASB networks
- *0* additional links are required for the reverse shuffle -- the existing links just need to be rewired.

The AWE switch, composed of 4x2 switching elements, requires 2*N* additional links:

- $\frac{N}{2}$ switching elements are added and each requires 4 inputs. This results in $4 * \frac{N}{2} = 2N$ additional links.

There are some advantages to using the AWE switch. Even though there are more new wires for the AWE switch, there is less total rewiring. When doubling the AWE switch, new links are simply added to the right of the existing switches. There is no need to disconnect the nodes from the existing switches. However, with the ASB switch extensive wiring of new and rewiring existing connections is required:

- *(N*/2 links): An *N*/2-chain of switching elements is formed.

- *(N* links): The *N*/2 nodes are disconnected from the existing ASB switches and connected to the new *N*/2-chain of switching elements.
- *(N* links): The output of the switching elements is connected using an inverse perfect shuffle to the inputs of the existing switches.
- *(N* links): The output of the *N*-ASB network is reconnected to different nodes using a reverse shuffle.

The AWE MultiRing switch focuses on maintaining the existing rings when expanding to include larger rings. This is advantageous when a ring of nodes are established and their sequence needs to be preserved. For example, consider a computer lab organized in a 4-node MultiRing with dedicated high-powered workstations used for real-time weather predictions and another computer lab organized in a 4-node MultiRing with household PCs used for maintaining household inventory. If these two MultiRings were combined, all of the computers could be used together to form a ring of 8 computers; yet the four high powered workstations need to remain connected when rings of 4 were created. The ASB MultiRing switch would require extensive rewiring to maintain the existing rings in the new network.

In Figure 4.35 and Figure 4.36, the different rings formed when combining existing networks into larger networks for AWE and ASB switches are provided. First a collection of 2-node switches are created. Then two 2-node switches are combined into 4-node networks. Finally, two 4-node networks are combined into one 8-node network. Notice in Figure 4.35 that ring configuration connecting *A, B, C* and *D* is available on both 4-AWE and 8-AWE switches. However, on the ASB switches in Figure 4.36 the ring connecting *A, B, C* and *D* is available on the 4-ASB switch but not on the 8-ASB switch.

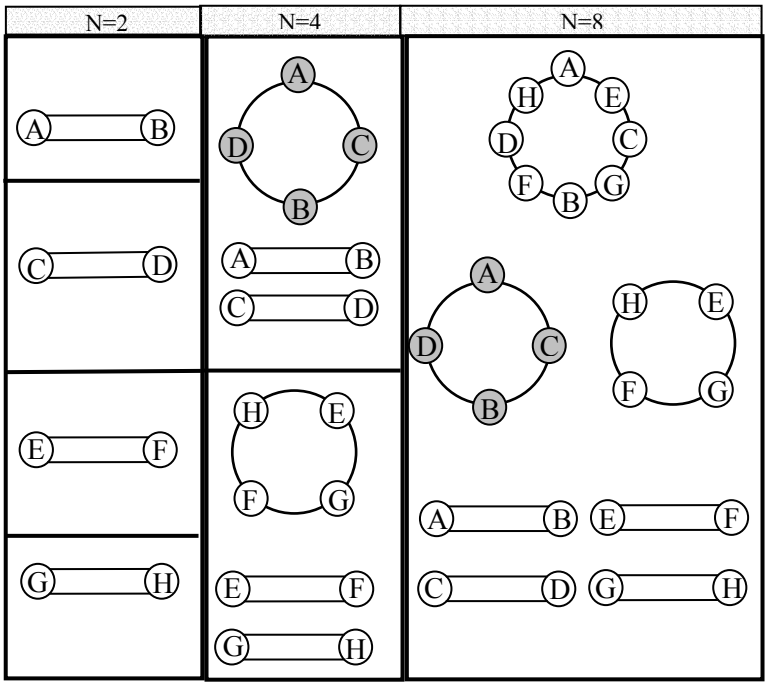


Figure 4.35: Expanding AWE switch

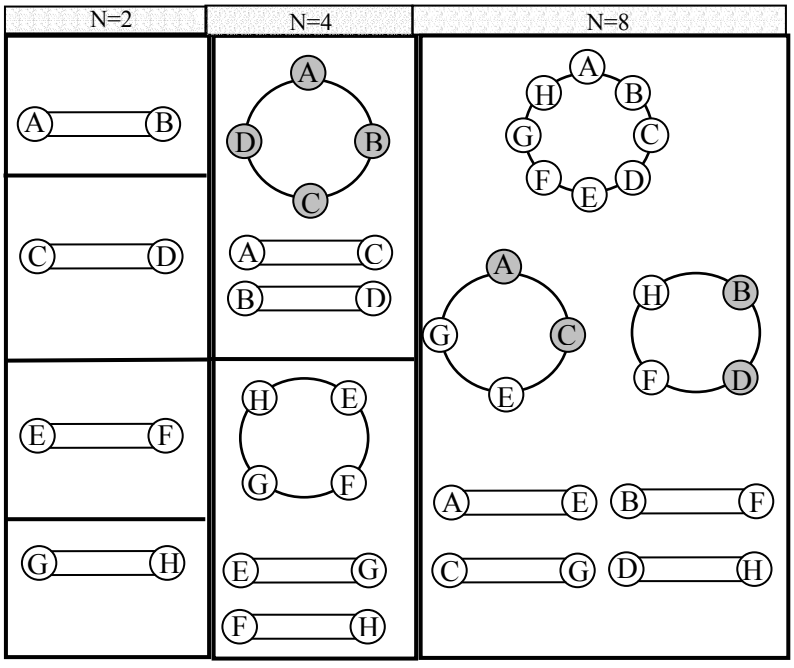


Figure 4.36: Expanding ASB switch

4.4 Unidirectional and Bidirectional Switches

The multi-stage interconnection networks have been illustrated with source nodes on the left and destination nodes on the right of the interconnection network. However, to implement the MultiRing, the source node *A* is the same as destination node *A*. To simplify the design of the switch, all input and output ports are rewired to appear on the left of the interconnection network. Transfer lines are added to each switching element to transfer the final output produced by the series of switches to the leftmost switch. Transfer lines connect all of the switching elements in a row. For example in Figure 4.37, the exchange elements on a 4-butterfly network are set to allow a clockwise ring of 4-nodes. The path from *A* to *B* is established by following the data lines (dark arrows) until the last column of switches and simply using the transfer lines (dotted arrows) to reach the destination node.

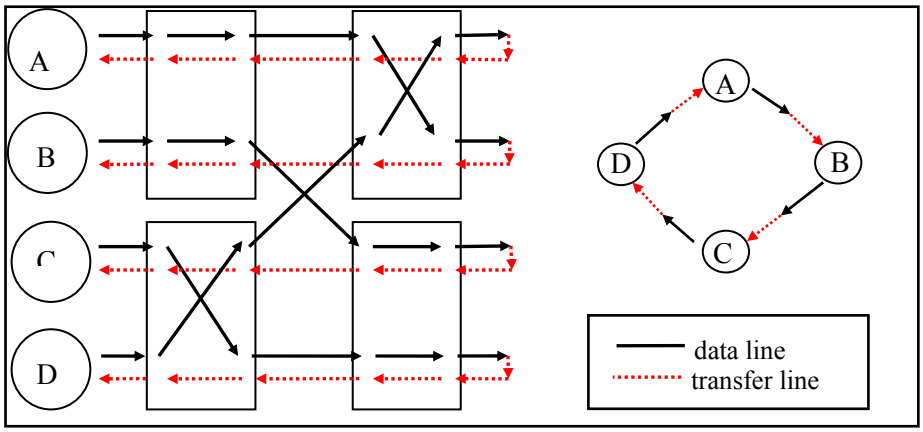


Figure 4.37: Transfer lines.

Unidirectional and bidirectional switches have both data and transfer lines. However in the unidirectional switch the data and transfer lines are unidirectional; the data lines always flow from left to right and the transfer lines always flow from right to left. In the fully bidirectional switch the data and transfer lines are bidirectional and can flow in both direction simultaneously. In the following sections unidirectional and bidirectional switches are discussed in detail.

4.4.1 Unidirectional Switches

Both MultiRing and AWE switches described in this chapter are unidirectional switches. The data on a ring flows in one direction for a configuration. Each of the nodes in a unidirectional MultiRing have one input port and one output port. In Figure 4.38 the input and output ports are depicted a) in the ring of 4 nodes and b) as nodes connected to a multi-state switching network.

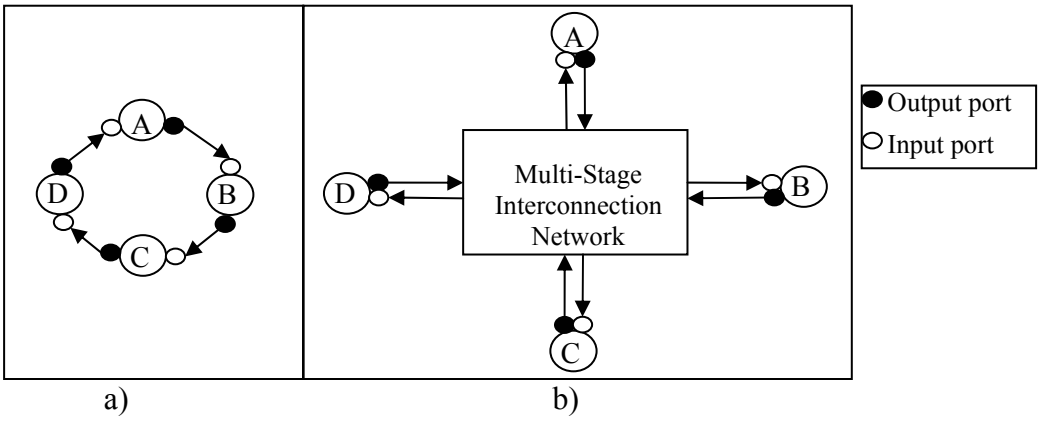


Figure 4.38: Input and Output ports in Ring

The flow of data on a ring can be either clockwise or counter-clockwise direction; but not at the same time. Different control words are required for clockwise and counter-clockwise directions. On ASB and AWE MultiRing switches, creating the counter-clockwise control word is easy. If control signal, C_i =1 for a clockwise ring, the counter-clockwise setting for the same ring requires C_j=1 for $j \leq i$. The counter-clockwise settings for 8 and 16 node switches are given in Figure 4.39.

MultiRing with 8 nodes		
Number of Nodes	Clockwise Controls	Counter-Clockwise Controls
	$C_2\,C_1\,C_0$	$C_2\,C_1\,C_0$
1	0 0 0	0 0 0
2	0 0 1	0 0 1
4	0 1 0	0 1 1
8	1 0 0	1 1 1

MultiRing with 16 nodes		
Number of Nodes	Clockwise Controls	Counter-Clockwise Controls
	$C_3\,C_2\,C_1\,C_0$	$C_3\,C_2\,C_1\,C_0$
1	0 0 0 0	0 0 0 0
2	0 0 0 1	0 0 0 1
4	0 0 1 0	0 0 1 1
8	0 1 0 0	0 1 1 1
16	1 0 0 0	1 1 1 1

Figure 4.39: Counter-clockwise controls for MultiRing switches

Two paths in the counter-clockwise configuration of 2 ring of 4 nodes are highlighted in the ASB switch, Figure 4.40, and in the AWE switch, Figure 4.41 . Two paths in the counter-clockwise ring of 8 nodes are highlighted in the ASB switch, Figure 4.42 and the AWE switch, Figure 4.43. The shaded elements represent different settings than for clockwise ring configurations.

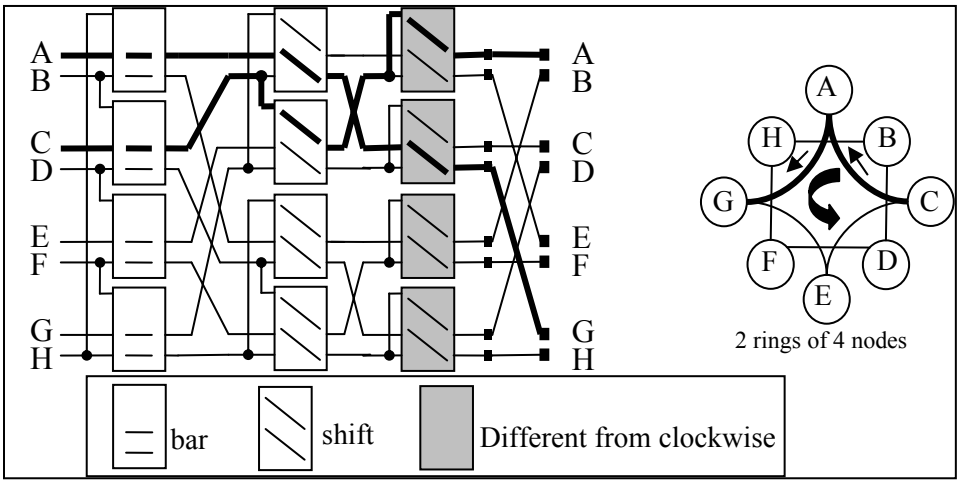


Figure 4.40: Two counter-clockwise rings of 4 nodes on 8-ASB switch

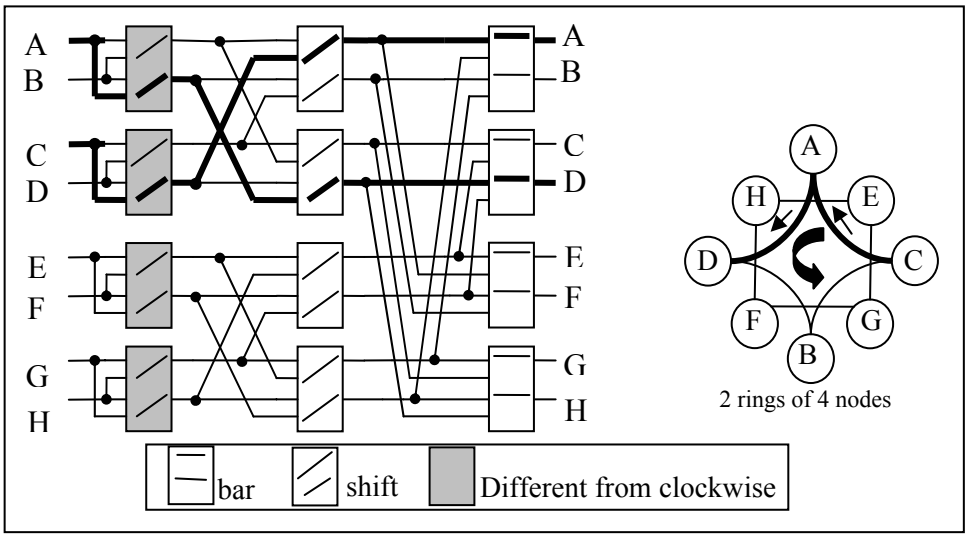


Figure 4.41: Two counter-clockwise rings of 4 nodes on 8-AWE switch

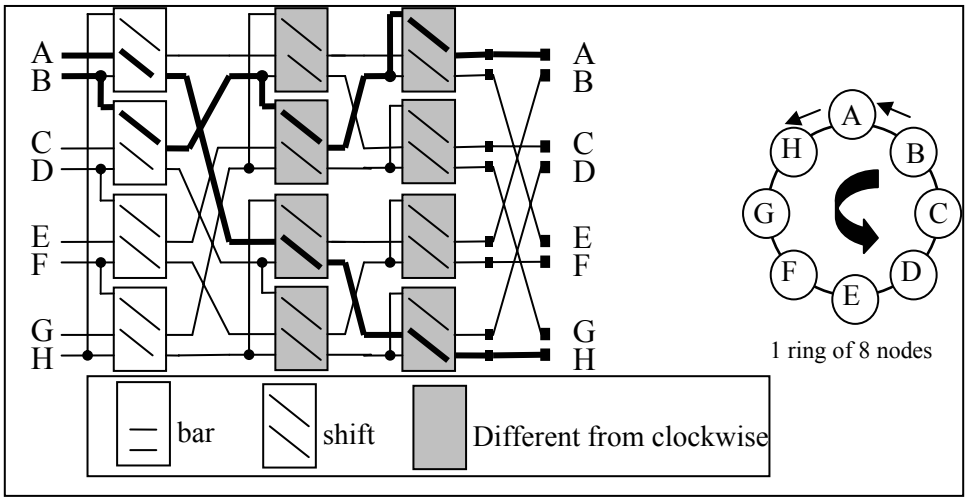


Figure 4.42: Counter-clockwise ring of 8 nodes on 8-ASB switch

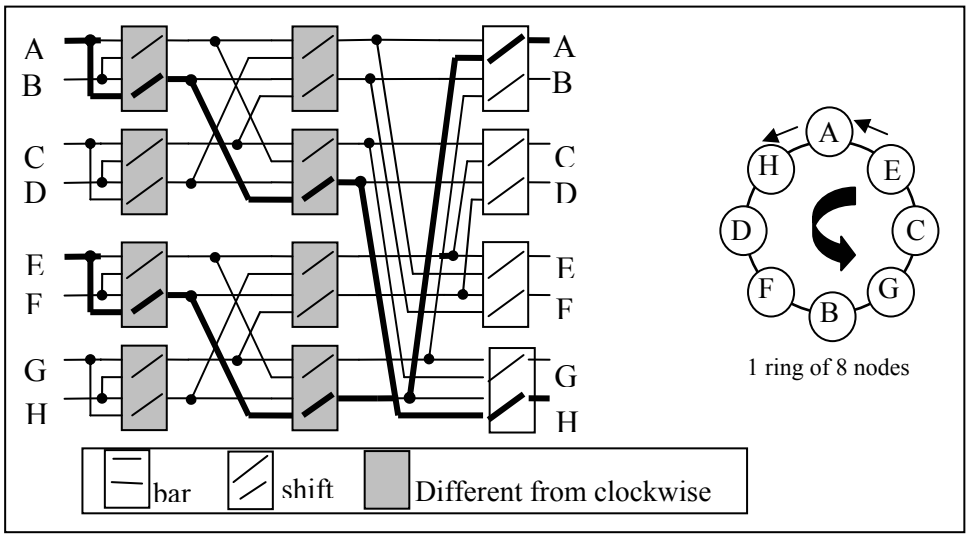


Figure 4.43: Counter-clockwise ring of 8 nodes on 8-AWE switch

4.4.2 Bidirectional Switches

A lot of applications need to use only one direction at a time for communication; in these applications a unidirectional switch is fine. However, in some applications there may be a need for a node to send data to two nodes simultaneously. This is a case for *bi-directional switches*.

There are two ways to implement bidirectional switches: a) use two unidirectional switches or b) create a fully bidirectional switch.

With two unidirectional switches, sending data to left and right neighbors on a MultiRing configuration is accomplished by having two rings with the same nodes available at the same time (see Figure 4.44). One ring is clockwise and the other ring is counter-clockwise. Use the clockwise ring to send data to right neighbor and receive from left. Use the counter clockwise ring to send data to the left neighbor and receive data from the right neighbor. One unidirectional switch always provides clockwise rings and the other unidirectional switch always provides counter-clockwise rings for the same configuration.

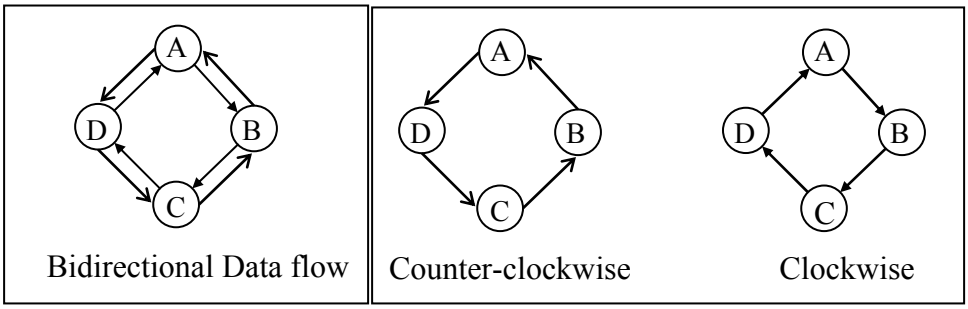


Figure 4.44: Two rings form bidirectional dataflow.

Refer to Figure 4.45 for the layout of the bidirectional switch constructed with two unidirectional switches. Notice the data and transfer lines are unidirectional.

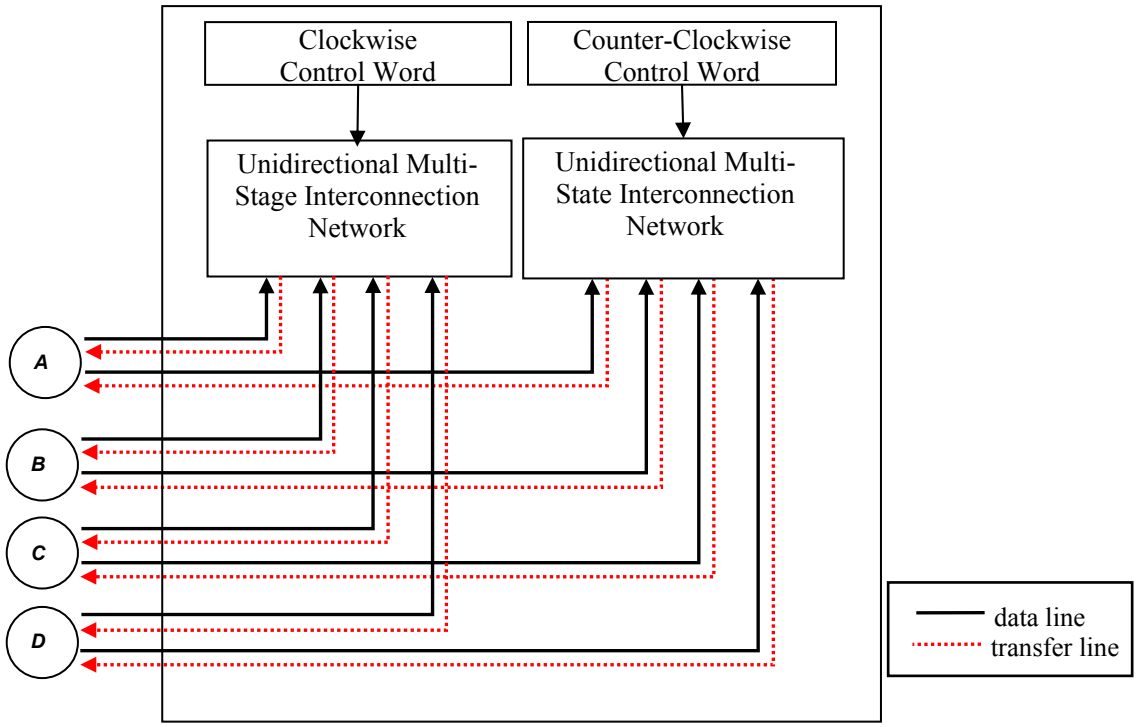


Figure 4.45: Constructing a Bidirectional Switch with Two Unidirectional Switches

Instead of using two unidirectional switches, another option is to create a larger bidirectional switch that contains bidirectional data and transfer lines (see Figure 4.46). There will not be a need to establish two different rings for clockwise and counter-clockwise data flow. To send data to the right neighbor (clockwise data flow), send data on the output port connected to the *data line* and receive data on the input port connected to *the transfer line*. To send data to the left neighbor (counter-clockwise data flow), send data on the output port connected to *the transfer line* and receive data on the input port connected to the *data line*.

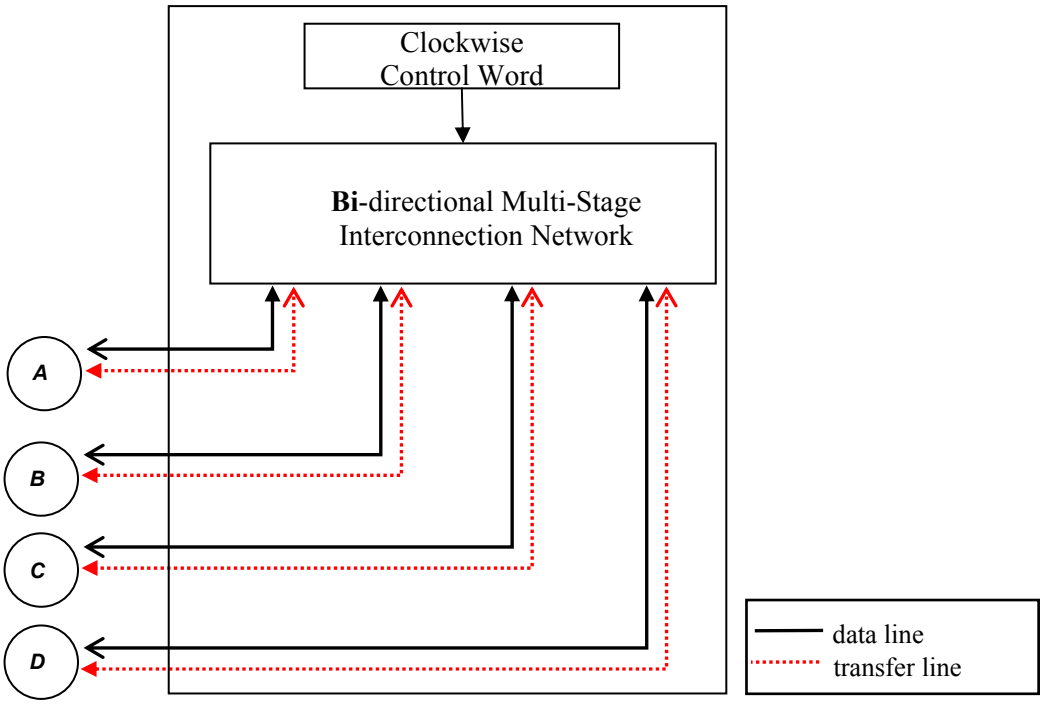


Figure 4.46: Bidirectional Switch

As an illustration, consider the bidirectional 4-butterfly in Figure 4.47. A node connects to its left neighbor using the bidirectional transfer lines and connects to its right neighbor using the bidirectional data lines.

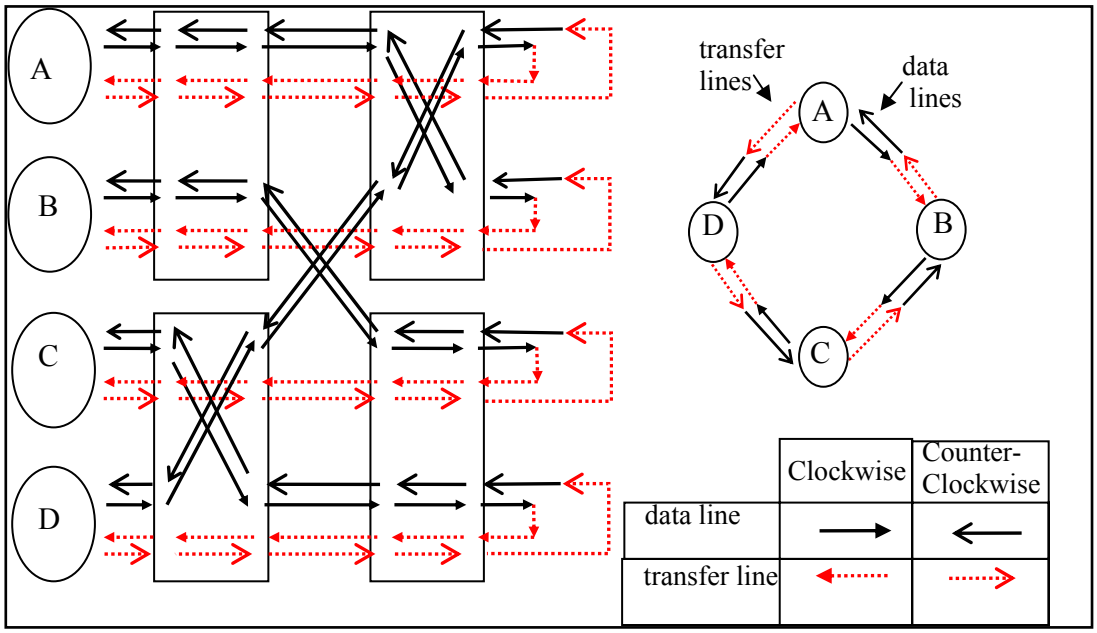


Figure 4.47: Bidirectional Data and Transfer Lines

For example, to establish counter-clockwise ring of 4 nodes, use the same control word as for clockwise (see Figure 4.37 for clockwise ring) but send data out on transfer line and receive data from the data line (see Figure 4.48)

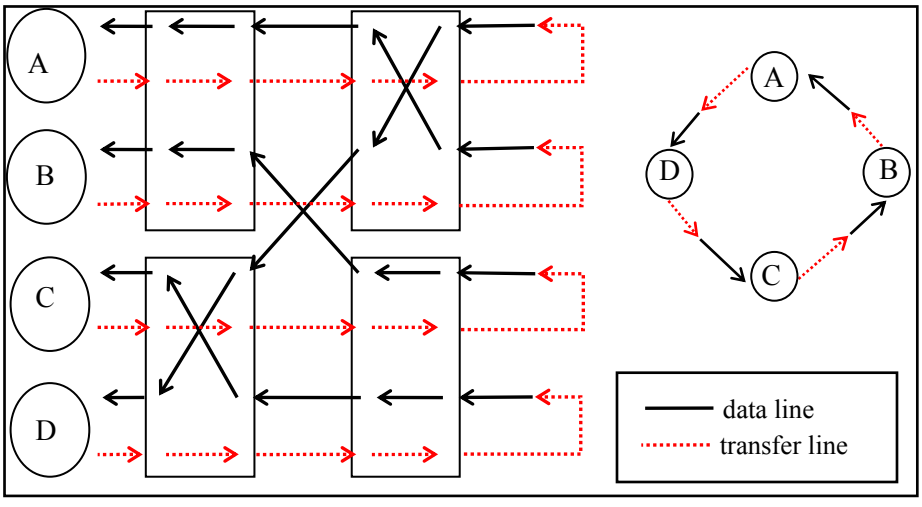


Figure 4.48: Counter-Clockwise Data flow on Bidirectional Switch

4.5 Switchboxes and Reconfiguration

In a MultiRing, each node will connect to a switchbox containing the multi-stage interconnection network and the switchbox will establish the connections for log N different configurations of the network. Each node will have 2(log N) possible connections, one for each neighbor, but only two connections will be active at one time. A main feature of the MultiRing is that multiple nodes can communicate simultaneously. In this section details of three types of reconfiguration algorithms are described: 1) automatic switch reconfiguration, 2) manual switch reconfiguration, and 3) smart switch.

4.5.1 Automatic Switch Reconfiguration

A MultiRing may be designed with a switch that automatically cycles through all through all of the configurations, starting with 2^0 rings of 2^r nodes down to 2^{r-1} rings of 2 nodes. The switch just forwards data along the communication channel to the neighboring node for a given configuration. It remains at a configuration for a set time to allow data to be completely delivered across the channel. The switch will remain open long enough for the sender to poll the receiver and receive an acknowledgement that it is ready to receive data in addition to the time necessary to send the message.

As an enhancement, the switch may maintain a cache of recently used configurations and increase the time it spends on the preferred configurations. Switching to a new configuration happens really quickly, so even if only one configuration is needed for an algorithm, the time spent on cycling through the unused configurations is negligible. With automatic switch reconfiguration, an algorithm designer can think of the network as fully connected and not worry about requiring a certain configuration.

4.5.2 Manual Switch Reconfiguration

A MultiRing may be designed with a switch that waits until all nodes agree on a certain configuration before reconfiguring the network. Most of the algorithms implemented on the MultiRing require the nodes to operate in barrier synchronized fashion so the other nodes that have data to send will eventually require the same configuration. Only after the switch receives *all* expected requests for a configuration, does the switch reconfigure the network.

A manual switch may also be useful when the MultiRing needs to maintain its configuration for a long time such as when implementing a ring of nodes as a pipeline where each node in the ring represents one processing phase. When one node is finished processing data, it will send its results to the next node in the ring. The configuration remains the same until all data has passed through all nodes in the ring.

4.5.3 Smart Switch

When a smart switch is used, each node just sends a message to the switch without waiting for a particular configuration. The switch maintains a composite routing table that specifies the required configurations necessary to establish communication between all nodes in the MultiRing. The switch reads the messages and determines the expected configuration and direction from the composite routing table. The network is reconfigured and the message is sent along the correct path

4.6 Concluding Remarks

This chapter presents the details of two multi-stage interconnection networks designed for the MultiRing. The original MultiRing switch, referred to as the ASB switch, proposed by Arabnia and Smith [12] provides point-to-point networking without contention. However, to

expand the network to include additional nodes requires disconnecting the existing nodes from the existing switch.

One of the goals of this research was to create an interconnection network that could expand with minimal cost. The AWE switch groups nodes into pairs and is able to build larger switches without disconnecting the nodes from the existing switch. Thus existing ring formations remain intact during the expansion of the network. The AWE switch also has the added benefit that now a MultiRing does not have to have exactly 2^r nodes. This means that fewer than N processors can be added to the network during the expansion of an N –node MultiRing.

This chapter also contains a description of how to create a bidirectional MultiRing and the controls necessary to provide clockwise and counter-clockwise ring communication. In addition, algorithmic issues regarding manual and automatic switches have been examined. With a manual switch, nodes issue a request for a configuration, whereas with an automatic switch, the switch automatically cycles through all of the configurations and remains at a configuration for a set time.

In the next chapter moves into a study of different communication models for message-passing on the MultiRing. The algorithms presented assume that the automatic switch is used; thus they do not contain explicit requests for a configuration as would be needed on a manual switch.

CHAPTER 5 : COMMUNICATION STRATEGIES

Many of the static networks described in Chapter 2 can be simulated on the MultiRing. All of the required links for the static network may not be available at the same time – only a subset of links may exist in a single configuration of the MultiRing, but all are available on one of the configurations. Thus, the MultiRing may have to be reconfigured several times to support the required communication for some algorithms, and algorithms that involve a subset of the communication links at a given time are the best candidates for the MultiRing.

Each configuration of a MultiRing with $N=2^r$ nodes is denoted by *config(N, c),* which represents a network with 2^{c-1} rings of 2^{r-c+1} nodes, where $1 \leq c \leq r+1$. As illustrated in Figure 5.1, the interconnects available on an 8-node MultiRing include *config(8, 1)* with 1 ring of 8 nodes, *config(8, 2)* with 2 rings of 4 nodes, *config(8, 3)* with 4 rings of 2 nodes, and *config(8, 4)* with 8 rings of 1 node. The *config(8,4)* is not addressed in any of the communication strategies presented in this chapter since it does not provide additional communication links.

Bhandarkar and Arabnia [15, 17] have proved basic definitions and proofs of MultiRing topology. In particular:

- All 2D mesh topologies of size 2^i x 2^j are subsets of a 2^r- node MultiRing where $i+j=r$.
- The hypercube with 2^r processors can embed all possible configurations of an 2^r-node MultiRing (though not simultaneously).
- Any algorithms which uses the communication links along a single dimension of a cube with 2^r nodes at any given point can be mapped to a 2^r-node MultiRing.

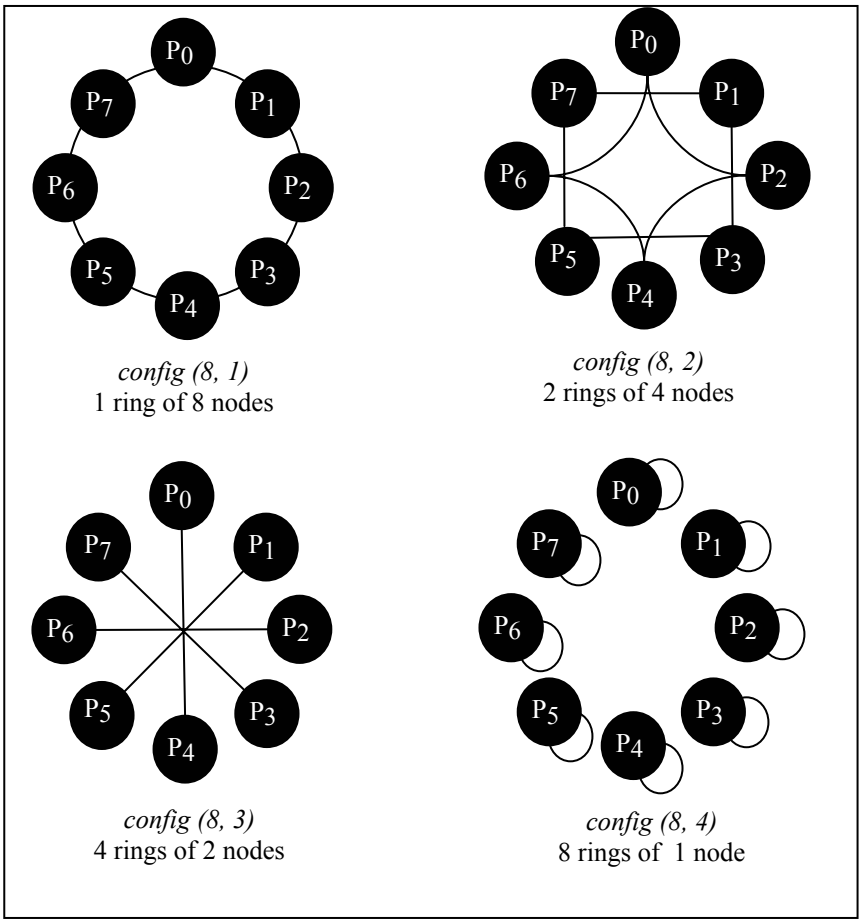


Figure 5.1: Configurations of an 8-node MultiRing.

A node can connect to at most two other nodes in a single configuration. On an *N*-node MultiRing with nodes labeled consecutively from P_0 to P_{N-1}, in *config(N, c)*, P_i for $1 \le i \le N-1$, connects to its left neighbor, P_{left}, and to its right neighbor, P_{right}, which are defined as follows:

$$P_{left} = P_{(i+N-2^{c-1}) \bmod N}$$

$$P_{right} = P_{(i+N+2^{c-1}) \bmod N}$$

The communication strategies described in this chapter are designed to support algorithms that can run on static networks that can be embedded in a MultiRing. If the number of nodes in a static network is less than the number of nodes available in the entire MultiRing

network, a configuration with a ring that best fits the size of the static network is chosen as the main configuration. The MultiRing may still have to be reconfigured, but every effort should be made to restrict communication to include only nodes in the main configuration. This will aid in multiprogramming. When groups of processors work on programs independently, a node not on the main configuration will not have to route messages between other nodes working on different programs. It may not be feasible to ensure this for all problems, but when possible, communication should be limited solely to nodes in the main configuration. This approach has not been implemented in previous papers on the MultiRing [8,17].

For instance, in an 8-node MultiRing, if a problem exists that requires only four nodes in a static network, then *config (8,2)* that has rings of 4 nodes will be selected as the main configuration. Nodes P_0, P_2, P_4, and P_6 are selected to work on the problem, leaving P_1, P_3, P_5, and P_7 to work on other problems. Communication will be restricted to nodes only in the main configuration. P_0 can connect to P_2, P_4 and P_6; it should not communicate with P_1, P_3, P_5 or P_7. Therefore, *config(8,2)* with 2 rings of 4 nodes and *config(8,3)* with 4 rings of 2 nodes can be used for communication, but *config(8, 1)* with 1 ring of 8 nodes will not be used to transmit data. To illustrate, the four nodes in the group P_0, P_2, P_4 and P_6 are highlighted gray in Figure 5.2. Notice that in *config (8,1)*, P_0 is linked to P_1 and P_7, which are not in its group. However, in *config(8,2)* and *config (8,3),* P_0 connects to nodes in its group. In general, once the main configuration, *config (N, mainConfig*), has been selected, only configurations *config (N, c),* where $mainConfig \leq c \leq r$, are needed to support all required communication between nodes.

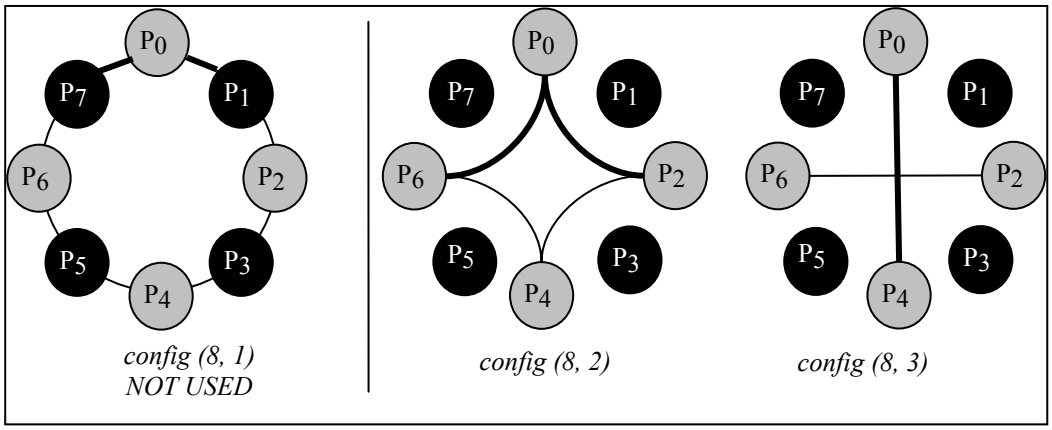


Figure 5.2: Communicating with nodes in main ring configuration.

This chapter provides different models for communication on the MultiRing. Each model includes strategies for choosing 1) the best interconnect configuration to simulate a network, 2) the method of reconfiguring the network to provide multiple independent sub-networks, and 3) input and output connections for an arbitrary node on the network. Section 5.1 describes the pipeline communication model, section 5.2 describes the n-cube communication model, section 5.3 describes the 2D grid communication model, and section 5.4 describes the complete binary tree communication model.

5.1 Pipeline Communication Model (PCM)

A linear array is an obvious static network that is embedded in the MultiRing, and a pipeline is one parallel strategy that can easily be implemented on a linear array. As data flows in one direction on a pipeline, it can be simulated on a *unidirectional* MultiRing. All of the links required by a pipeline are available in the same ring configuration. As a result, once the initial configuration is established for the pipeline, the MultiRing does not need to be reconfigured.

To establish a pipeline of L nodes on a MultiRing with $N=2^r$ nodes, one should reconfigure the network into the main configuration, *config* (N, *pipeConfig*), where $pipeConfig = r - \lceil \log L \rceil + 1$. If the number of nodes required in a pipeline does not exactly

match the number of nodes on a ring, using this formula allows establishment of the smallest ring that can support the pipeline. What is the best configuration to form a pipeline differs according to how many nodes are in the MultiRing. For example, to establish a pipeline with 3 nodes on an 8-node MultiRing, one should choose *config (8, 2)* with 2 rings of 4 nodes.

$$pipeConfig = 3 - \lceil \log 3 \rceil + 1$$

$$pipeConfig = 3 - 2 + 1$$

$$pipeConfig = 2$$

However, to establish the same 3 node pipeline on a 16-node MultiRing, one should choose *config (16, 3)* which has 4 rings of 4 nodes.

$$pipeConfig = 4 - \lceil \log 3 \rceil + 1$$

$$pipeConfig = 4 - 2 + 1$$

$$pipeConfig = 3$$

Reconfiguring the MultiRing to the smallest ring needed to establish the pipeline supports the goal of multi-programming. On some configurations, more than one pipeline can operate independently. In fact, on *config (N, c)*, there can be 2^{c-1} independent pipelines, each with 2^{r-c+1} nodes for $1 \leq c \leq r$. As illustrated in Figure 5.3, an 8-node MultiRing supports 1 pipeline of 8 nodes, 2 pipelines of 4 nodes and 4 pipelines of 2 nodes. With 1 pipeline, P_0 is the head; with 2 pipelines, P_0 and P_1 are the heads, and with 4 pipelines, P_0, P_1, P_2, and P_3 are the heads of the pipelines. In general, the heads of the pipelines on *config (N, c)* are P_i for $0 \leq i < 2^{c-1}$.

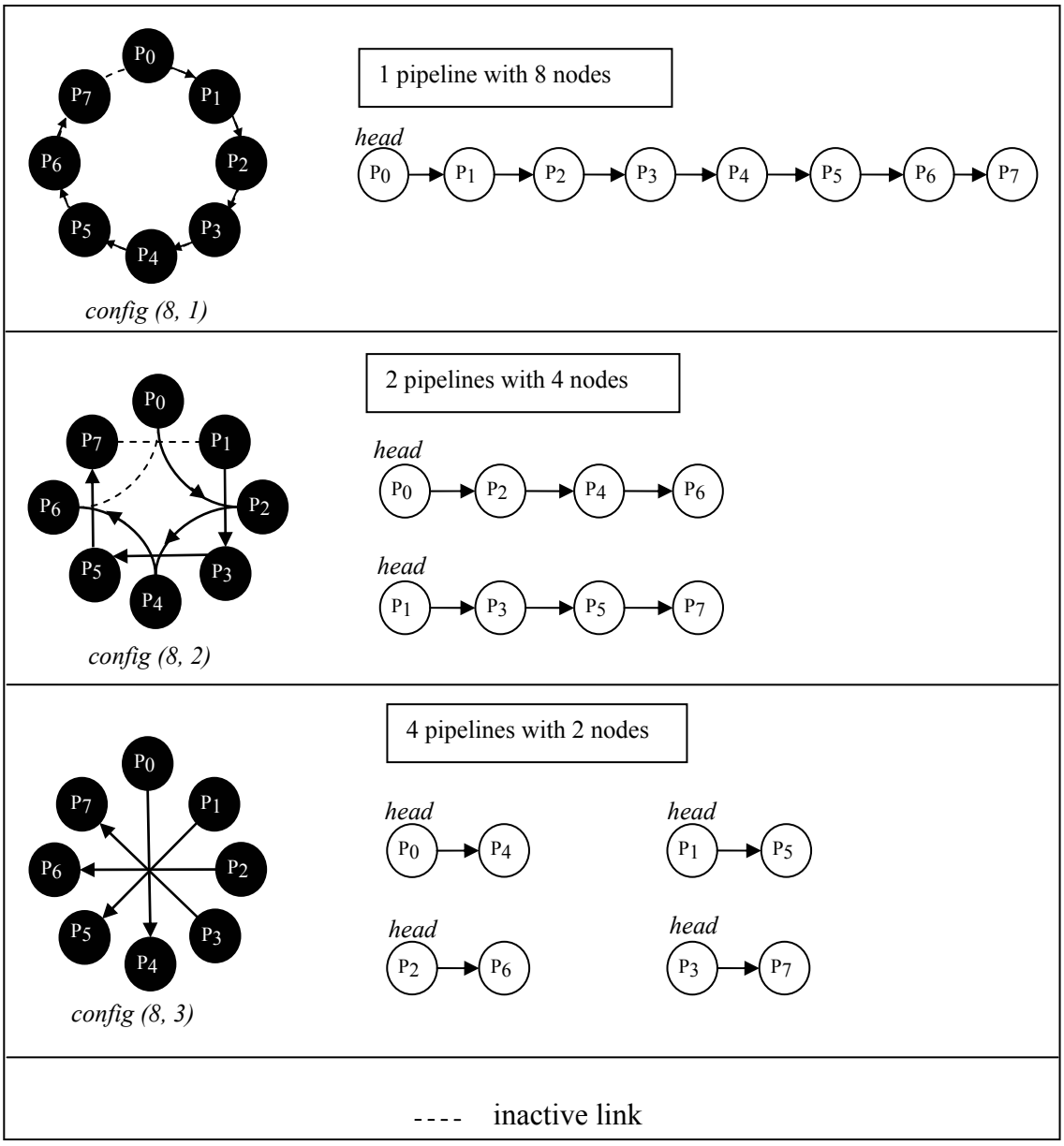


Figure 5.3: Simulating a Pipeline

Since the pipeline is unidirectional, each node has only one input, $P_{in,}$ and one output, P_{out}. An arbitrary, P_i, receives data from its left neighbor and sends data to its right neighbor. In an N-node MultiRing for $1 \leq pipeConfig \leq r$, P_{in} and P_{out} for P_i are defined as follows:

$P_{out} = P_{right}$ on *config(N, pipeConfig)*

$P_{in} = P_{left}$ on *config(N, pipeConfig)*

5.2 Cube Communication Model

As described in Chapter 2, each node in a hypercube with $N=2^r$ nodes has an r-bit ID string. The bits are arranged in a sequence 123…r, where the first bit represents the 1st dimension, the second bit represents the 2nd dimension and so on until the r^{th} bit represents the r^{th} dimension. Two nodes are connected in a hypercube if their IDs only differ by 1 bit. For example, in a hypercube with 8 nodes, node `000` connects with node `100` in the 1st dimension, `010` in the 2nd dimension and `001` in the 3rd dimension.

A *bidirectional* MultiRing can simulate a hypercube. There is a direct mapping between the node IDs on the hypercube and the binary representations of the MultiRing node IDs. An 8-node hypercube and the corresponding MultiRing node IDs are illustrated in Figure 5.4.

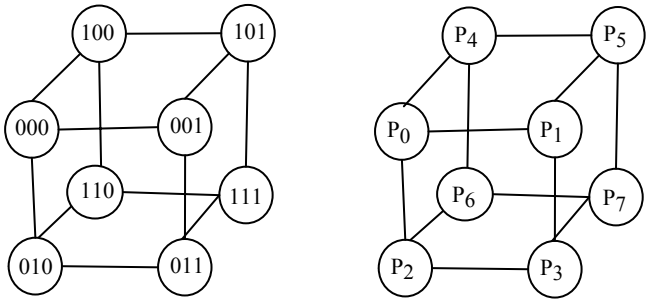


Figure 5.4: Using the MultiRing node IDs on a hypercube.

Bhandarkar and Arabnia [17] proved that algorithms that only use one dimension of the hypercube at a time can be simulated on the MultiRing. Each configuration of the MultiRing can provide communication for nodes connected in a single dimension. In general, communication for dimension d is established on *config (N, dConfig)*, where $dConfig = r + 1 - d$.

Figure 5.5 illustrates the links available on the corresponding configuration of the MultiRing for the first, second and third dimensions of an 8-node hypercube. In dimension 1-D,

where P_0 (**0**00) connects with P_4 (**1**00), *config(8,3)* is required (i.e. *dConfig = 3 + 1 – 1 = 3).* In dimension 2-D, where P_0(0**0**0) connects with P_2 (0**1**0), *config (8,2)* is required (i.e. *dConfig = 3 + 1 – 2=2).* In dimension 3-D, where P_0 (00**0**) connects with P_1 (00**1**), *config (8,1)* is required (i.e. *dConfig = 3 + 1 –3 = 1).*

The number of nodes in the MultiRing network affects *dConfig* for a given dimension. For example, on an 8-node MultiRing, 1-D communication is established on *config (8, 3)* (i.e. *dConfig* = 3 + 1 – 1 = 3) and 3-D communication is established on *config (8, 1)* (i.e. *dConfig* = 3 + 1 – 3 = 1). However, on a 16-node MultiRing, 1-D communication is established on *config (16, 4)* (i.e. *dConfig* = 4 + 1 – 1 = 4), and 3-D communication is established on *config (16, 2)* (i.e. *dConfig* = 4 + 1 – 3 = 2).

Hypercubes of a smaller dimension than *r-D* can be simulated in an a 2^r-node MultiRing by calculating the configuration for its highest dimension, *highConfig*, and using only MultiRing configurations between *config (N, highConfig)* and *config(N, r)*. For example, to simulate a 2-D hypercube on a MultiRing with 8 nodes, *highConfig* is the configuration which links nodes in the 2-D dimension, *config(8, 2);* and only *config (8, 2)* and *config (8, 3),* which link nodes in the 1-D and 2-D dimensions, are needed to provide all communication links necessary for the 2-D hypercube.

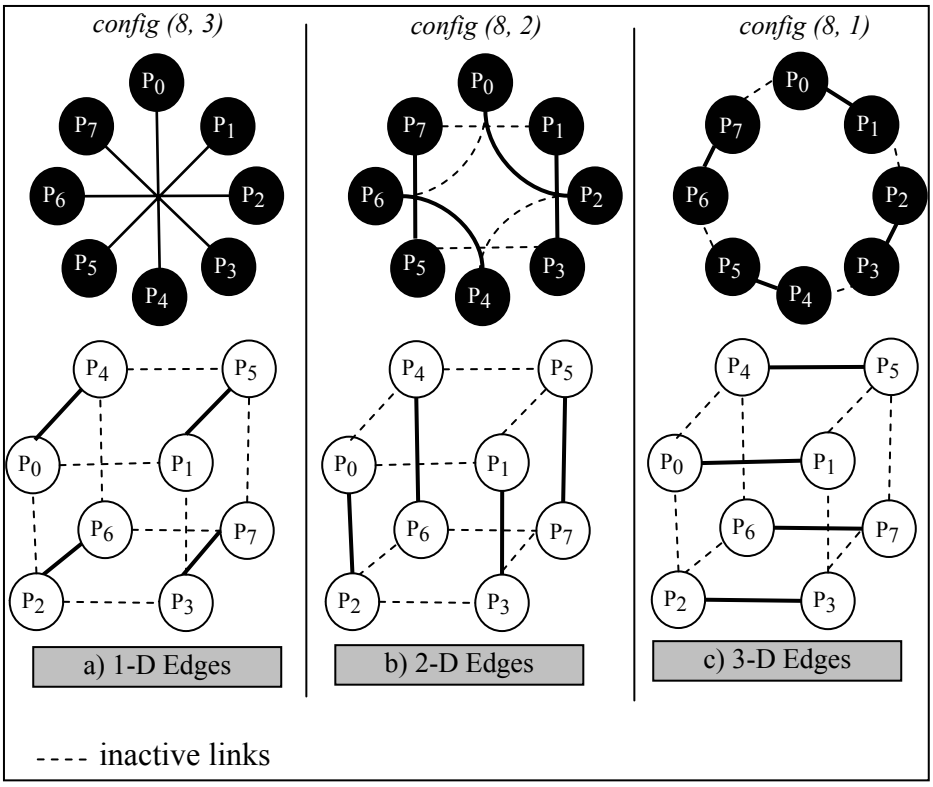


Figure 5.5: Simulating a hypercube on a MultiRing.

As described in Chapter 2, a 3-D hypercube can be separated into two 2-D hypercubes by ignoring the 3-D links. These two 2-D hypercubes can be simulated independently on a MultiRing. Figure 5.6 illustrates a 3-D hypercube with the corresponding MultiRing node IDs and the *highConfig* configuration. (The dashed lines are the 3-D edges.) *Config (8, 2)*with two rings of 4 nodes establishes the *highConfig* for both independent hypercubes; one ring has P_0, P_2, P_4, P_6. and the other ring has P_1, P_3, P_5, P_7.

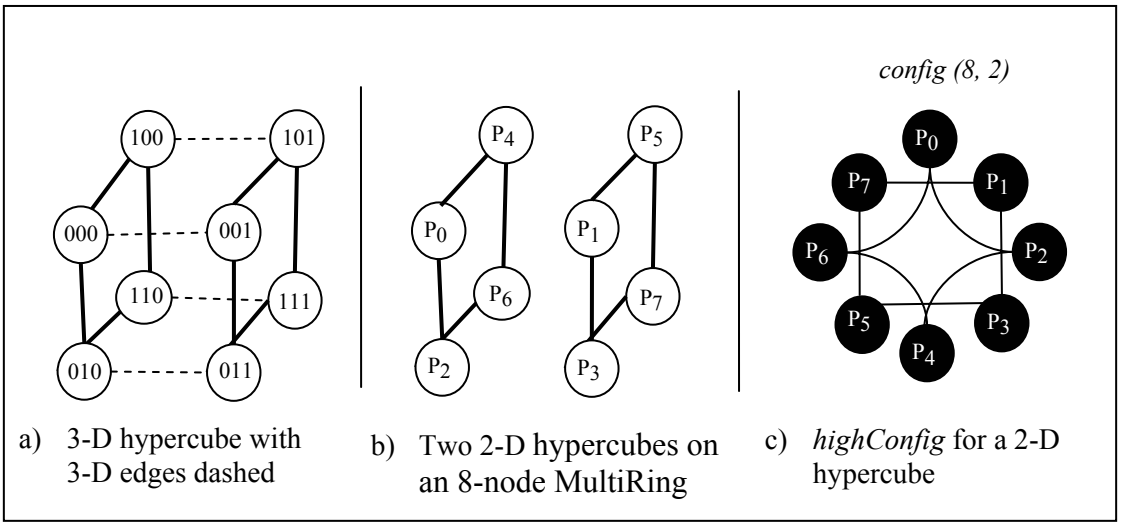


Figure 5.6: Mapping two 2-D hypercubes on a MultiRing

Reconfiguring the MultiRing to the smallest ring needed to establish the hypercube aids in the goal of multi-programming. On some configurations, more than one hypercube can operate independently. In fact, on *config (N, c),* where *c=highConfig*, there can be 2^{c-1} independent hypercubes each with dimension (*r-c+1)*.

Table 5.1 provides the number of independent hypercubes on various configurations of 8- and 16-node MultiRings.

Table 5.1: Independent hypercubes on 8- and 16- node MultiRings

N	*highConfig*	Number of Hypercubes	Dimension
8	3	4	1
8	2	2	2
8	1	1	3
16	4	8	1
16	3	4	2
16	2	2	3
16	1	1	4

Even though a *bidirectional* MultiRing is used to simulate a hypercube, each node only needs one input, P_{in} and one output, P_{out}. Node P_i communicates with node P_j on dimension d if only their dth bits of their IDs are different. They both send and receive data from each other. The larger node in the pair sends data to its left neighbor and the smaller node sends data to its right neighbor. For example, on dimension 2, P_0 connects with P_2; P_2 sends data to its left neighbor, P_0, and P_0 sends data to its right neighbor, P_2.

A node can determine if it is the larger node of the pair by examining the dth bit of the binary representation of its node ID. If it is a 1, this node is the larger node. P_{out} and P_{in} are defined as follows:

If P_i is the larger node:

$$P_{out} = P_{left}$$

$$P_{in} = P_{right}$$

Else

$$P_{out} = P_{right}$$

$$P_{in} = P_{left}$$

5.3 Grid Communication Model (GCM)

An R x C grid is a two-dimensional mesh with R rows and C columns. It can be embedded in an N-node MultiRing when $(R * C) \leq N$. Each cell in the grid can be mapped to a different node on the MultiRing. A cell at position (row, col) where $0 \leq row < R$, $0 \leq col < C$ and $(R * C) \leq N$ is mapped to a node P_i where $i = \left(row * 2^{r-\lceil \log R \rceil}\right) + col$. When 1x16, 2x8, 4x4, 8x2, and 16x1 grids, each with 16 cells, are mapped onto a 16-node MultiRing, the corresponding node IDs are arranged in row-major form (Figure 5.7).

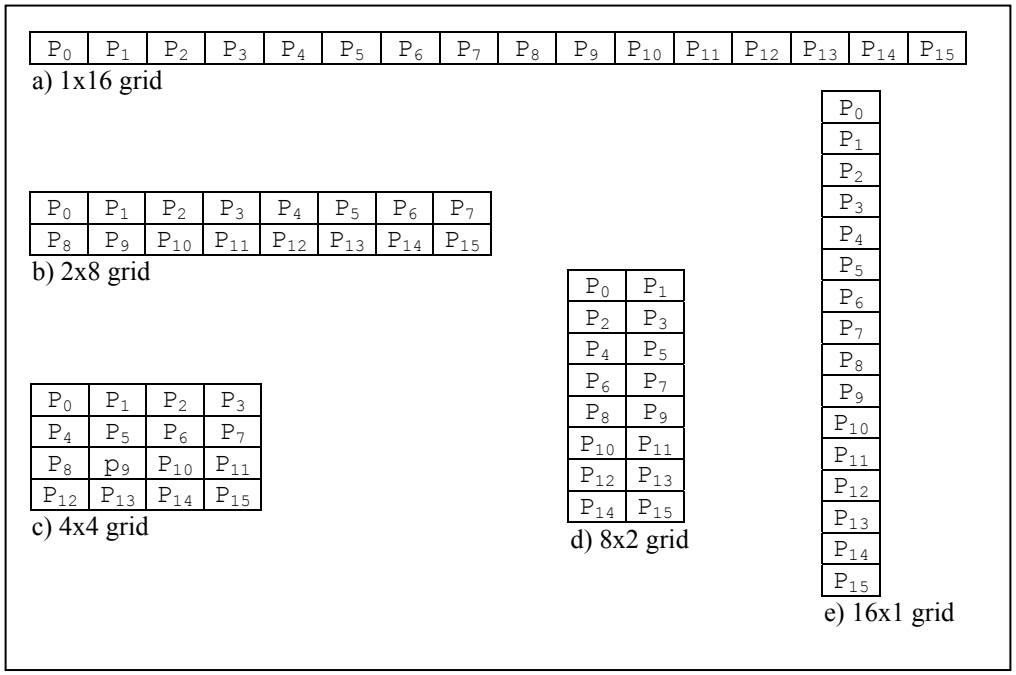


Figure 5.7: Grids with 16 cells.

Only two different MultiRing configurations are needed to provide all communication links between a node and its four nearest neighbors, P_{north}, P_{south}, P_{east} and P_{west}. *Config (N, 1*) with a single ring of all nodes establishes communication across rows on the grid, thereby providing links to P_{east} and P_{west} neighbors. *Config (N, colConfig)* with separate rings for each column, establishes communication on columns of the grid, providing links to P_{north} and P_{south}, where $colConfig = r + 1 - \lceil \log R \rceil$. For example, a 8x2 grid on a 16-node MultiRing requires *config(16,1)* for row communication and *config(16,2)* for column configuration (Figure 5.8). On the other hand, a 4x4 grid requires *config(16,1)* for row communication and *config(16,3*) for column communication (Figure 5.9).

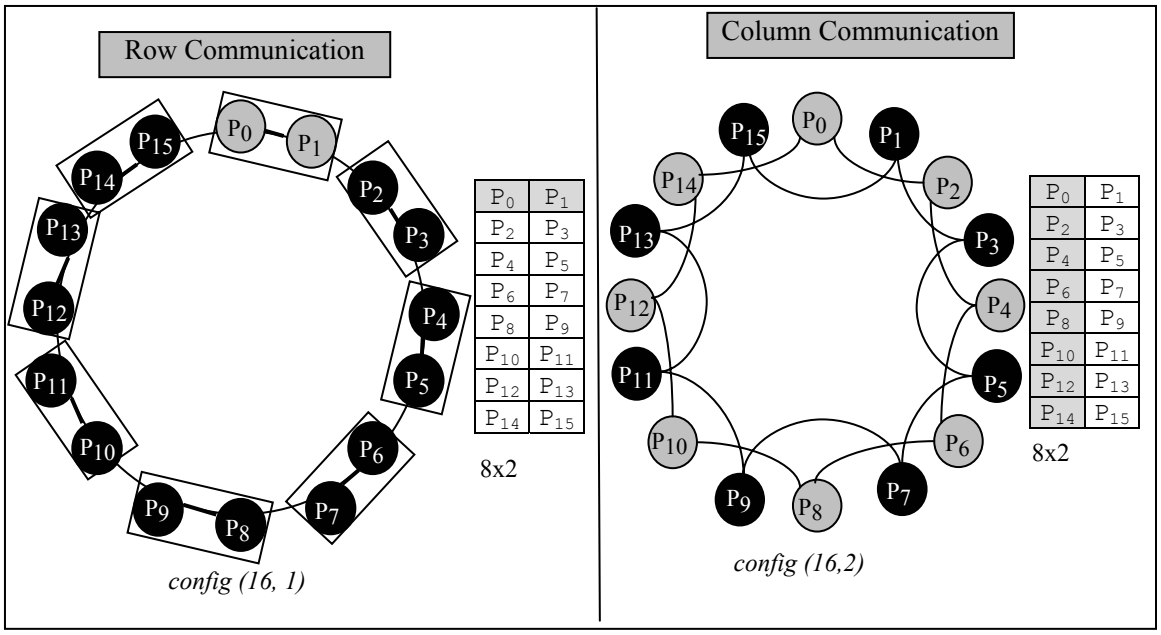


Figure 5.8: Mapping a 2 x 8 grid on a MultiRing

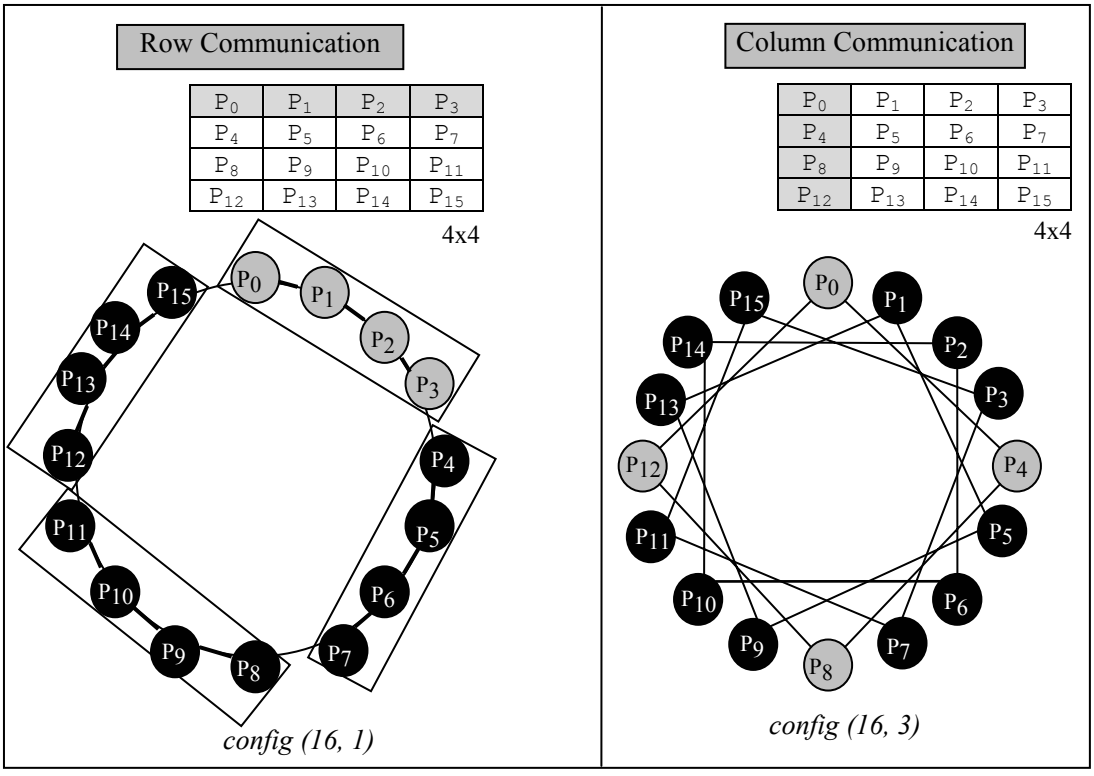


Figure 5.9: Mapping a 4 x 4 grid on a MultiRing

If the number of cells in a grid is less than the number of nodes in the MultiRing (i.e. *(R * C) < N*) some nodes will not be used during the mapping. For this reason, the numbering of nodes will not appear sequential. If a 3x3 grid, a subset of a 4x4 grid, is mapped on a 16-node MultiRing, *config (16 , 1*) and *config (16, 3)* provide links for row and column communication respectively. P_0, P_1 and P_2 represent the cells on the first row, P_4, P_5 and P_6 represent the cells on the second row, and P_8, P_9 and P_{10} represent the cells on the third row (Figure 5.10).

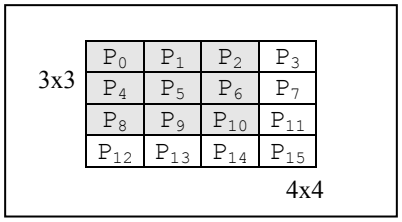


Figure 5.10: 3x3 grid is a subset of a 4x4 grid.

Different strategies could be used to map a grid with fewer cells than nodes available on the MultiRing. For example a 2x2 grid is a subset of 2x8, 4x4 and 8x2 grids, each with 16 cells (Figure 5.11). The formula presented here for computing the *colConfig* always finds the configuration with the smallest number of nodes on a ring that can represent the grid for column communication. On a 16-node MultiRing, the 2x2 grid is considered a subset of the 2x8 grid, and 8 rings with 2 nodes are used for column communication.

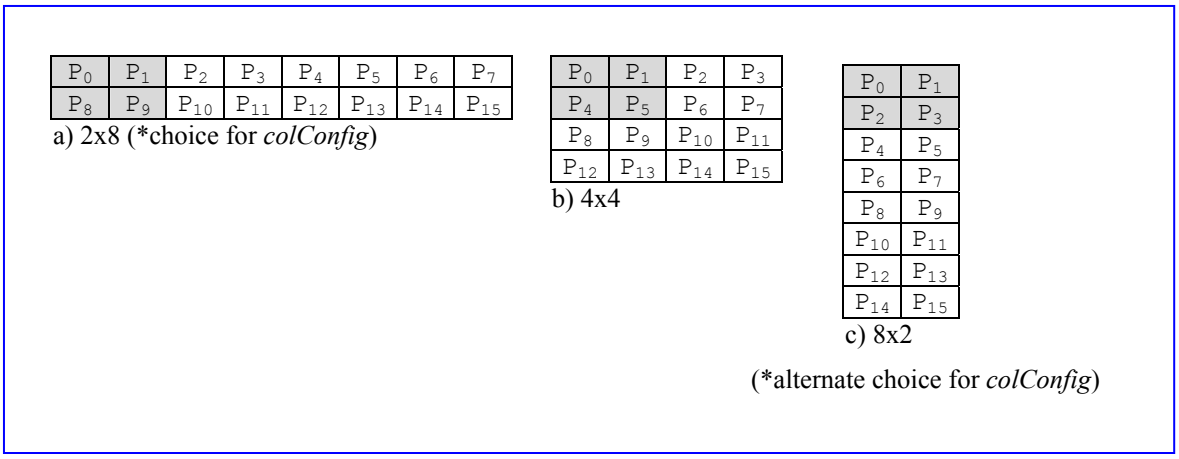


Figure 5.11: Embedding a 2x2 grid in different grids with 16 nodes.

An alternative formula for computing *colConfig* could be used which 'best fits' the number of columns in the grid instead of the number of rows (i.e., $colConfig = 1 + \lceil \log C \rceil$). The formula for mapping a cell to a node also changes; a cell at *(row, col)* maps to P_i where $i = \left(row * 2^{\lceil \log C \rceil}\right) + col$ for $0 \le row < R$, $0 \le col < C$ and $(R*C) \le N$. In this case, a 2x2 grid is considered a subset of the 8x2 grid which requires rings of 8 nodes for column communication.

$P_{i,}$ an arbitrary node on an *R x C* grid, communicates with its north, south, east and west neighbors as follows:

$P_{north} = P_{left}$ on *config (N, colConfig)*

$P_{south} = P_{right}$ on *config (N, colConfig)*

$P_{west} = P_{left}$ on *config (N, 1)*

$P_{east} = P_{right}$ on *config (N, 1)*

Without torus connections, boundary nodes do not have all four neighbors. $P_{i,}$ can determine if it is a boundary node in *R x C* grid with a few calculations:

P_i is on the left border if ((*i* % *MaxCol*) = 0)

P_i is on the right border if ((*i* % *MaxCol*) = *C-1*)

P_i is on the top border if ($i < MaxCol$)

P_i is on the bottom border if ($i > ((R\text{-}1)*MaxCol)$)

where $MaxCol = 2^{r-\lceil \log R \rceil}$.

5.4 Tree Communication Model (TCM)

A complete binary tree is a tree in which all the leaves have the same depth and all internal nodes have a degree of 2. When one is mapping a tree with *N-1* nodes onto an *N*-node MultiRing, the corresponding nodes on the MultiRing are determined by numbering the nodes in the tree from 1 to *N-1* using an in-order traversal. The one extra node, P_0 connects to the root of the complete binary tree.

The MultiRing must be reconfigured to support all communication links to simulate a tree; only communication at a single level of the tree is available on each configuration. On an *N*-node MultiRing, a node at height *h* communicates with its children on *config (N, childConfig)* and with its parent at *config (N, parentConfig)*, where *parentConfig* = *childConfig*+1, *childConfig* = $h + r - (TreeHeight + 1)$, and *TreeHeight* is the height of the complete binary tree. In Figure 5.12, a complete binary tree with 7 nodes is mapped onto an 8-node MultiRing. In *config (8, 3)*, P_0 connects to the root, P_4, and the nodes on the tree are numbered from P_1 to P_7 using an in-order traversal. P_4 connects to its children, P_2 and P_6 on *config(8, 2)*. P_2 and P_6 connect to their children on *config (8, 1)*.

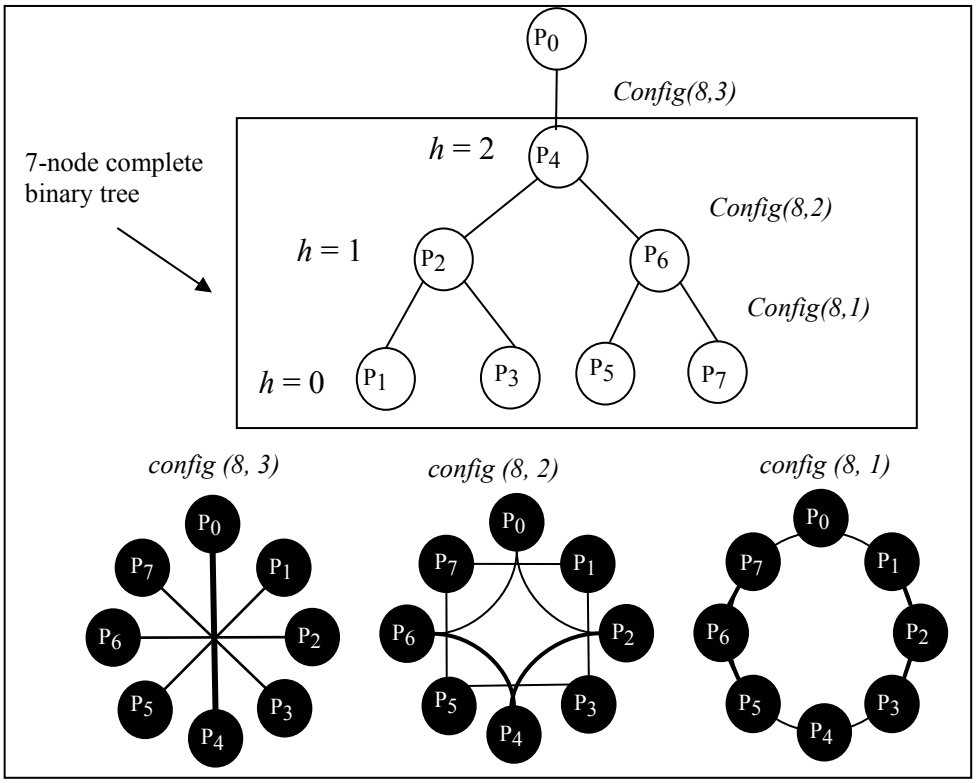


Figure 5.12: Simulating a complete binary tree with 7 nodes on an 8-node MultiRing.

Multiple independent complete binary trees can be simulated on the MultiRing. *Config (N, c)* with 2^{c-1} rings of 2^{r-c+1} nodes has 2^{c-1} independent complete binary trees with 2^{r-c+1}-1 nodes. Node P_i, where $i<2^{c-1}$ connects with the root of the complete binary tree in *config (N, r).* For example, two independent complete binary trees with 7 nodes are embedded in a 16 node MultiRing (Figure 5.13) In *config (16, 2)* with 2 rings of 8 nodes, each ring contains all of the nodes in one tree. P_0, P_2, P_4, P_6, P_8, P_{10} , P_{12} and P_{14} are associated with one tree and P_1, P_3, P_5, P_7, P_9, P_{11}, P_{13} and P_{15} are associated with the other tree. P_0 connects to P_8, the root of one tree, and P_1 connects to P_9, the root of the other tree on *config(16, 4).*

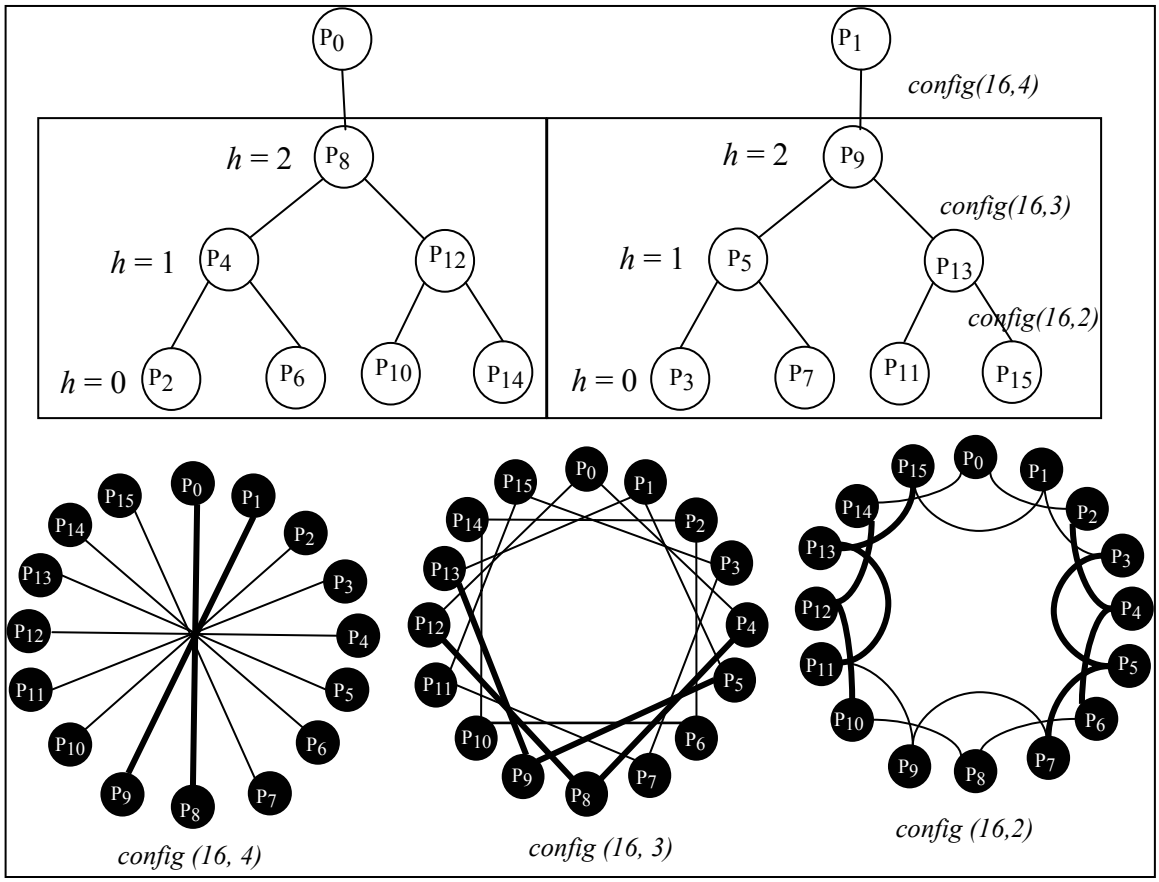


Figure 5.13: Two 7-node complete binary trees in 16-node MultiRing.

The leaf nodes are at height h=0 and by definition do not have links to any child nodes. Therefore one should consider P_i as an arbitrary node on the complete binary tree at height h where $1 \leq h \leq (r\text{-}1)$. Node P_i at height h contains links to its right child, $P_{rightChild}$ and its left child, $P_{leftChild}$ where:

$P_{rightChild} = P_{right}$ on *config (N, childConfig)*

$P_{leftChild} = P_{left}$ on *config (N, childConfig)*

To determine the parent, one must first identify if a node is a right child or left child. The right child on *config(N, c)* has a 1 in the c^{th} bit of the binary representation of its node ID. Thus, P_i is the right node if $(i \,\%2^{c+1}) > 2^c$.

If P_i is the right child,

$P_{parent} = P_{left}$ on *config(N, parentConfig)*

Else if P_i is the left child

$P_{parent} = P_{right}$ on *config(N, parent Config*)

A node is the root if *parentConfig* = *r*.

5.5 Concluding Remarks

This chapter has included communication strategies for supporting algorithms originally designed to run on static networks for implementation on the MultiRing. Algorithms developed for pipeline, n-cubes, 2D grids and trees can be implemented on the MultiRing; however, only a subset of links required by the algorithm may exist in a single configuration of the MultiRing. Thus, the MultiRing may have to be reconfigured several times during the execution of a program to provide the required communication links.

The creation of unidirectional and bidirectional MultiRing switches was a focus of Chapter 4. Pipelining algorithms can be implemented on a unidirectional MultiRing in which all nodes can send data to the right (left) neighbor during a configuration. However, n-cube, grid and tree algorithms are best implemented on a bidirectional MultiRing because all nodes may send data to both left and/or right neighbor during a configuration.

Multiprogramming on a network of workstations is one of the key motivators of this research and, it is in an effort to support multiprogramming; suggestions for determining the MultiRing configuration with a ring that best fits the number of nodes needed in the program have been given. An automatic switch that cycles through all configurations can quickly reconfigure the MultiRing to allow multiple independent sub-networks. The communication

strategies developed ensure that a node in a given sub-network will not have to route messages between other nodes working on different programs.

The next chapter provides a collection of primitive communication-routing algorithms which can be used to direct the flow of messages on the MultiRing. Different algorithms will be presented that use the pipeline, cube, and tree communication models. It will also be shown that it is good to have a collection of communication models because, depending on the operation, one communication model may yield a simpler algorithm.

CHAPTER 6 : ROUTING OPERATIONS

In a parallel system, it is best for each node to have a direct link with all other nodes; however, as discussed in Chapter 2, this is not always feasible. The MultiRing has a limited set of direct links (*2*log N* -1), and routing algorithms must be developed to direct the flow of messages enabling each message to reach its destination with the fewest 'hops.' To be effective in a parallel system that allows multiprogramming, messages must be routed only between nodes which are working together on a program independently from other nodes.

Routing algorithms are developed based on the types of network they are intended for. On a token ring, a node reads a message from the network and forwards the message to the next node in the ring if the destination address is different than its address. On a bus, a node reads a message and simply ignores it or discards it if the destination address is different than its address. The routing and message-passing algorithms for the MultiRing must be designed for the reconfigurable nature of the MultiRing.

As discussed in Chapter 4, both unidirectional and bidirectional MultiRing switches can be used in a MultiRing network. If a unidirectional MultiRing switch is used, and data flows in a clockwise direction for all configurations of an 8-node MultiRing, P_0 can send data directly to P_1, P_2, and P_4, and it can receive data from P_7, P_5 and P_4. However, if a bidirectional MultiRing switch is used, and data flows in both clockwise and counter clockwise directions for all configurations on an 8-node MultiRing, P_0 can send and receive data directly to/from P_1, P_7, P_2, P_5 and P_4.

The direct connections are constantly changing with an automatic switch so that when a node receives a message destined for another node, it must decide on the configuration to which it will forward the message. What happens, for example, if P_0 has a message for P_3? There is not a configuration that provides a direct link between P_0 and P_3. Figure 6.1 illustrates three alternatives by which the message may reach P_3. These include:

a. P_0 sends the message to P_1 in *config(8, 1)* and P_1 forwards the message to P_3 in *config(8,2).* This uses only a RIGHT_LINK to forward the message in only 2 hops.

b. P_0 sends the message to P_4 in *config(8, 3).* P_4 forwards the message to P_6 in *config(8,2).* P_6 forwards the message to P_7 in *config(8,1).* P_7 forwards the message to P_3 in *config(8,3).* This strategy also uses only RIGHT_LINKS to forward messages, but this requires 4 hops.

c. P_0 sends the message to P_7 in *config(8, 1).* P_7 forwards the message to P_3 in *config(8,3).* This strategy requires both LEFT and RIGHT links to forward the message in 2 hops.

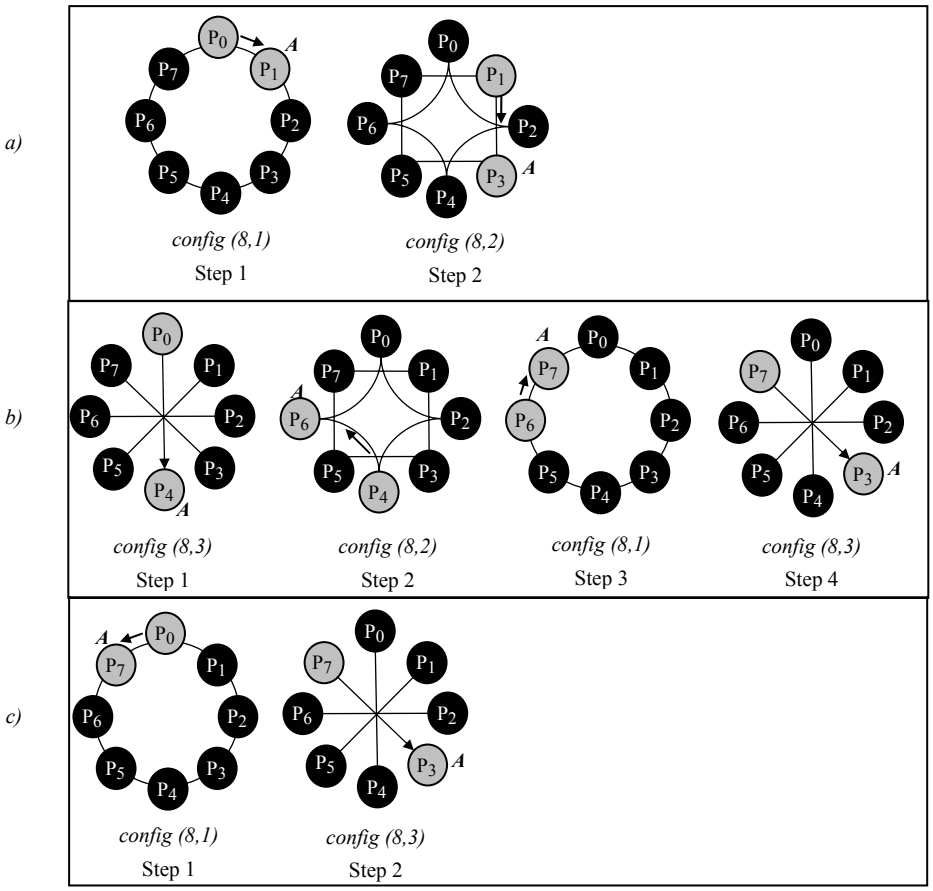


Figure 6.1: Alternatives for sending a message from P_0 to P_3.

In this chapter different routing algorithms are developed using the pipeline, cube and tree communication models presented in Chapter 5. All of the algorithms described will require at most *O (log N)* different configurations to transmit messages between nodes, but, depending on the operation, one communication model may yield a simpler algorithm.

If the *pipeline* communication model is used, messages are always routed in a clockwise direction for all ring configurations. A node can only send messages to its right neighbor. A node reads data from P_{left} and sends data to P_{right}. Figure 6.2 illustrates the data flow for the configurations of an 8-node unidirectional MultiRing network. In the third configuration, two links are depicted as one composite link.

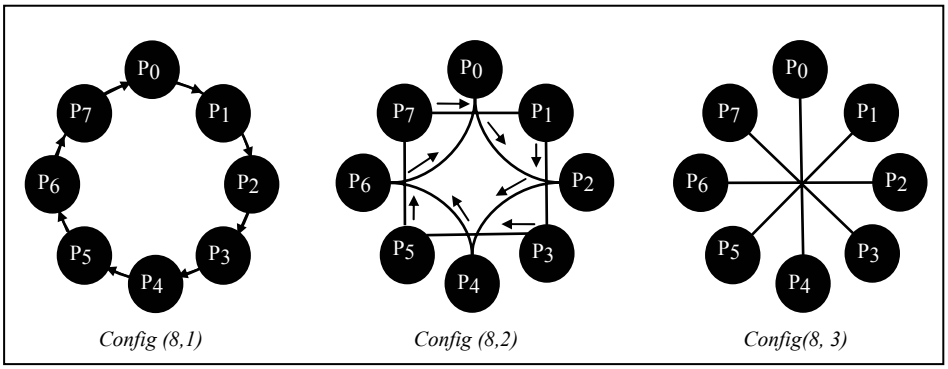


Figure 6.2: Pipeline communication on a MultiRing

When the *cube* communication model is used, each ring configuration establishes links between pairs of nodes that are connected in a single dimension of an *N*-node hypercube. A node can send data to only one neighbor, but, depending on the ring configuration (i.e. dimension edge), this may be either the left or right neighbor. Since nodes are connected in pairs on each ring on configurations *config (N, 1)* to *config (N, r-1*), there are links available on the MultiRing that will not be used. In Figure 6.3, the communication patterns for the configurations of an 8-node MultiRing are illustrated. The solid links represent active links and the dashed lines represent inactive links.

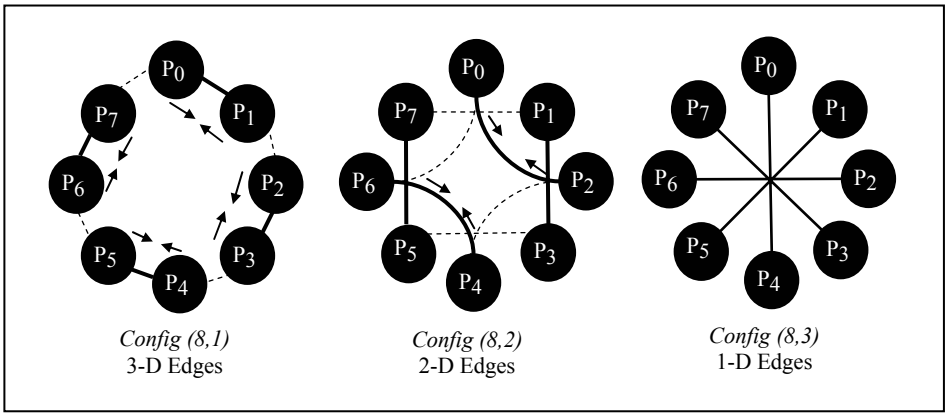


Figure 6.3: Cube communication on a MultiRing

A node in a tree network can send data to both left and right children. If the *tree* communication model is used on the MultiRing, nodes can send data to both left and right neighbors on a ring. In Figure 6.4, the data flow from node P_0 is provided for each configuration of an 8-node MultiRing that uses the tree communication model. P_0 can send data to P_7 and P_1 in *config(8,1)*, P_6 and P_2 in *config (8,2)* and P_4 in *config(8,3)*.

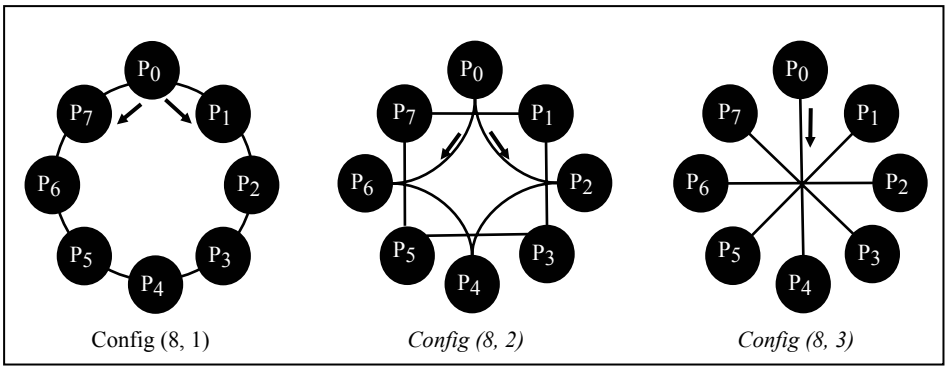


Figure 6.4: Tree communication on a MultiRing

Section 6.1 describes how to determine the configuration and link direction required to transmit messages using pipeline, cube and tree communication models. In sections 6.2 - 6.5, several routing algorithms for message-passing operations are described. They are as follows:

(6.2): *Arbitrary Send* – sending a message to another node in the MultiRing

(6.3): *Ring Broadcasting* – sending the same message to all nodes on a given ring

(6.4): *Group Broadcasting* – sending the same message to all nodes in a group of consecutive numbered nodes

(6.5): *Distributing* – sending different portions of a large message to different nodes on the ring.

The Arbitrary Send is really the only message-passing operation that is required, as all other operations can be written based on the Arbitrary Send. However, implementing special routines for broadcasting and distributing can be more efficient. For each message-passing

operation presented, multiple algorithms are provided to illustrate pipeline, cube and tree communication strategies.

6.1 Routing Messages on a MultiRing

A node must determine the correct configuration to use to send a message to its neighbor to ensure it will be delivered in *O (log N)* transfers. For instance, if P_0 wants to send a message to P_4, it should wait until the third configuration of the network when there is a direct link to P_4. If P_0 sends the message meant for P_4 to P_1 during the first configuration, it may take longer than *O(log N)* transfers for the message to reach P_4. In addition to determining the best configuration to send the message, the node must also determine the link direction.

The function **ConfigNumber**() utilized by each node in the network for creating a routing table that stores the 'best' configurations for sending messages is described in section 6.1.1. A function **LinkType**() that determines the link, LEFT_LINK or RIGHT_LINK, to use on a given configuration is presented in section 6.1.2. Finally in section 6.1.3 **PRight**(), **PLeft**() and **NextNode**() are provided that can determine the ID of the destination node on a given configuration.

6.1.1 Creating a Routing Table

Each node maintains a routing table that determines the first possible configuration it should use to send a message for all nodes in the network. A node's routing table has *N* entries where each entry is a configuration number. A zero, 0, indicates that the destination of the message is the same as the current node's ID, so the message has reached its destination. Table 6.1 provides a composite of all the routing tables for each node in an 8-node MultiRing. In actuality, each node stores just one column of the table - not the entire table. Each entry

represents the appropriate configuration number to send a mesage from the source node to reach its destination in *O(log N)* steps.

It is interesting to note that the initial configuration required to send a message from one node to an arbitrary node is identical for pipeline, cube and tree communication models. Therefore, the same algorithm can be used to create the routing table regardless of the type of MultiRing network.

Table 6.1: Routing table for an 8-node MultiRing

$\mathbf{P}_{destination}$ \ $\mathbf{P}_i$	$\mathbf{P_0}$	$\mathbf{P_1}$	$\mathbf{P_2}$	$\mathbf{P_3}$	$\mathbf{P_4}$	$\mathbf{P_5}$	$\mathbf{P_6}$	$\mathbf{P_7}$
P_0	0	1	2	1	3	1	2	1
P_1	1	0	1	2	1	3	1	2
P_2	2	1	0	1	2	1	3	1
P_3	1	2	1	0	1	2	1	3
P_4	3	1	2	1	0	1	2	1
P_5	1	3	1	2	1	0	1	2
P_6	2	1	3	1	2	1	0	1
P_7	1	2	1	3	1	2	1	0

To find the configuration required to send a message from P_i to P_j for $i \neq j$ and $0 \leq i, j \leq N-1$:

1. Calculate the distance from P_i.

   distance = (j > i) ? j – i : N + j – i;

2. Choose *config (N, c)* where the rightmost one in the binary representation of *distance* is at position *c-1*.

```
int   ConfigNumber (NODEID Pi, NODEID Pj)
{
      int c;
```

```
            distance = (j > i) ? j- I : N + j - i;
            c=0;
            //loop while there is not a 1 in the c^th bit of binary representation
            While ( !(distance & 1<<c))   // &-bitwise AND, <<-shift left
                  c++;
            c++; //increment c by 1
            Return c;
      }
```

A node can calculate ahead of time the configurations required to send a message to all other nodes and store these calculations in a routing table. However, although having a routing table saves time during message routing, it also takes up space. If there is not enough local memory to store the entire routing table, a node can compute the required configuration for a specific destination each time it sends a message. Space permitting, a cache of the most recently used destinations may be kept to save computation time.

6.1.2 Determining Direction of Links

A node has two links, one to its left neighbor and one to its right neighbor. After the configuration to use to send the message has been decided, it is necessary to determine which link the message should be outputted to, LEFT_LINK or RIGHT_LINK. The strategy for determining the link direction depends on the communication model.

a) Pipeline Communication Model

With the pipeline communication model, determination of the link direction is straight forward –always use the *right* link to send data. By definition, the direction of data flow on a pipeline remains the same for all configurations.

```
LINK LinkType(int config)
{
      LINK OUTPUT_LINK;
      OUTPUT_LINK = RIGHT_LINK
      Return OUTPUT_LINK;
}
```

b) Cube Communication Model

With the cube communication model, nodes are grouped in pairs and connections are created between nodes that are linked on a hypercube that represent a single dimension per each configuration. If two nodes are connected in a configuration, the node with the larger ID uses the left link to send data and the node with the smaller ID uses the right link to send data. To determine if the node, P_i, is the larger node in the pair on *config(N, c)*, one must examine the c^{th} bit of the binary representation of P_i's ID. If P_i has a one in the c^{th} bit, data is sent to the left neighbor, otherwise data is sent to the right neighbor.

```
LINK LinkType(int config)
{     LINK OUPUT_LINK;
      int c = config;
      // If Pi has the larger id of the pair, the cth bit is a 1
      If (Pi & (1<<(c-1))) //&-bitwise AND, <<-Shift left c-1 times
            OUTPUT_LINK = LEFT_LINK
      Else
            OUTPUT_LINK = RIGHT_LINK
      Return OUTPUT_LINK;
}
```

c) Tree Communication Model

With the tree communication model, a node may send data out of either its left or its right link for a given configuration. The linear distance from source node determines the link. One must check the location of the destination node in relation to the source node. If the destination node is one of the *N/2* nodes to the right of the source node, the right links are used, otherwise the left links are used. For example in Figure 6.5a) P_0 uses the *right* link to send messages to nodes P_1, P_2, P_3 and P_4 and the *left* link to send messages to the other nodes, P_7, P_6, and P_5. In Figure 6.5b) P_5 uses the *right* link to send messages only to nodes P_6, P_7, P_0 and P_1.

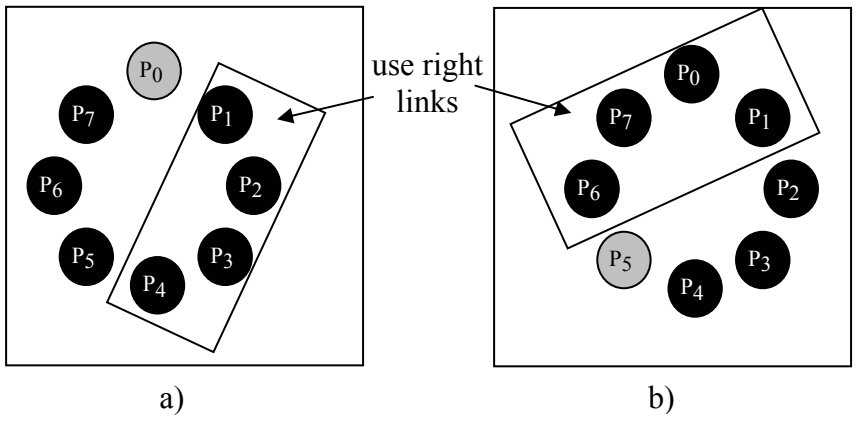


Figure 6.5: Link direction on Tree MultiRing.

The algorithm for determining which link to use is given below:

```
LINK LinkType (NODEID Pdest)
{
      LINK OUTPUT_LINK;
      If (Pdest < Pi && ((Pi-Pdest)>= N/2) ||
         (Pdest > Pi && ((Pdest-Pi)>= N/2)
         OUTPUT_LINK = RIGHT_LINK
```

```
      Else
         OUTPUT_LINK = LEFT_LINK
      Return OUTPUT_LINK;
}
```

6.1.3 *Determining the Next Node on the Ring*

Functions can be created to determine the ID of the next node on a ring for a given configuration. **PLeft**() and **PRight**() use the fact that on configuration *c*, the next node P_j for node P_i is determined by P_i's ID:

$$j = ((i + N) \pm 2^{c-1}) \bmod N$$

```
NODEID PLeft (int config)
{
      Return (Pi + N – pow(2,config-1)) %N;
}

NODEID PRight (int config)
{
      Return (Pi + N + pow(2,config-1)) %N;
}
```

The next node on a ring is determined by the link direction and the communication model used. The next node for a pipeline is always **PRight**(). The next node for using the either the tree or cube communication model is **PRight**() on the right link and **PLeft()** on the left link.

```
NODEID NextNode (int config, LINK link, COMMUNICATION_MODEL CM)
{     NODEID Pnext;
      if (CM == PIPELINE)
            Pnext = PRight(config);
      Else  If (CM == TREE || CM == CUBE)
            If (link == RIGHT_LINK)
                  Pnext = PRight(config);
            Else
```

```
                    Pnext = PLeft(config);
        Return Pnext;
}
```

6.1.4 Proofs

Defnition of MultiRing

Arabnia and Bhandarkar have defined and N-node MultiRing in [8] and [17]. In an *N*-node MultiRing where $N=2^r$, the nodes are numbered from P_0 to P_{N-1} and there are *r+1* different configurations. Each configuration is denoted by *config (N, c)* which represents a network with 2^{c-1} rings of 2^{r-c+1} nodes where $1 \leq c \leq (r + 1)$. Each node on a ring contains two links to connect to the adjacent nodes in the ring. In *config (N, c),* P_i contains a link to its right neighbor P_{Right} and its left neighbor P_{Left} where:

$$Right = (i + N + 2^{c-1}) \bmod N$$

$$Left = (i + N - 2^{c-1}) \bmod N$$

Definition of distance between two nodes

Consider a clockwise unidirectional ring of *N* nodes which is formed in *config (N, 1)* of an *N*-node MultiRing. Observe that the distance from P_i to P_j can be calculated by counting the number of links between P_i and P_j by starting with the right neighbor of P_i and proceeding in a clockwise direction until P_j.

$$Distance = (j - i + N) \bmod N$$

Case 1: There is not a "wrap around" ($j \geq i$)

$$Distance = j - i$$
$$= (j - i + N) \bmod N$$

Case 2: There is a wrap around (j < i). The total number of links is determined in three steps:

The distance from P_i to P_{N-1} = (N-1) - i

The distance from P_{N-1} to 0 = 1

The distance from P_0 to P_j = j

Distance = (j - i + N) mod N

Distance = ((N-1) - i) + 1 + j
= N - 1 - i + 1 + j
= N - i + j (since j < i this value is not > N)
= (j - i + N) mod N

The distance between P_i and P_j ranges from 0 to N-1 and can be represented in binary notation in *log N*, r, bits where the bits are numbered from 1 to r (*Binary* (distance) = $b_r \ldots b_2 b_1$).

It follows that if routing a message from P_i to P_j involves examining each bit of the distance only once, this routing method will finish in *log N* steps.

Definition of Pipeline Communication

In the pipeline communication model, messages are always routed in a clockwise direction. Thus, a node can only send a message to its right neighbor on the ring.

Theorem: Routing a message from P_i to P_j in an *N*-node MultiRing where $N = 2^r$ and $0 \leq i, j \leq (N-1)$ using pipeline communication can be accomplished in *O (log N)* steps.

Proof: Routing using pipeline communication involves changing each 1-bit of the distance to a 0-bit. When all bits are 0 the message will have reached its destination, P_j.

Base Case: If $P_i = P_j$, distance = 0 and all bits are 0. Thus, the message has reached its destination.

General Case: If $P_i \neq P_j$, distance $\neq 0$ and there is at least 1 bit set to 1.

Let *c* refer to the rightmost 1-bit, where $1 \leq c \leq r$. In pipeline communication on *config (N, c),* P_i sends data to its right neighbor P_{right}. This reduces the remaining distance by 2^{c-1}. As a result bit *c* is set to 0.

$P_{Right} = (i + N + 2^{c-1}) \bmod N$ (by definition of MultiRing)

Distance from P_i to P_j:

Distance = (j − i + N) mod N (by definition of distance)

Distance from P_{Right} to P_j in terms of i

$$\begin{aligned}\text{Distance} &= (j - ((i + N + 2^{c-1}) \bmod N) + N) \bmod N \\ &= (j - ((i + N) \bmod N + 2^{c-1}) + N) \bmod N \\ &= (j - ((i + N) \bmod N) - 2^{c-1} + N) \bmod N \\ &= (j - ((i + N) \bmod N) + N) \bmod N - 2^{c-1} \\ &= (j - i + N) \bmod N - 2^{c-1} \\ &= (\text{Distance from Pi to Pj}) - 2^{c-1}\end{aligned}$$

P_{Right} calculates the remaining distance; now the distance bits numbered from 1 to *c* are all 0. P_{Right} examines the remaining bits (from *c+1* to *r*) for the rightmost 1-bit. At the rightmost 1-bit, P_{Right} will forward the message and the total distance will be reduced. This process continue until all 1-bits have been examined in the distance and distance = 0. Thus the message will reach P_j in *log N* bits.

Call Statement: `Route (N, i, j, 1);`

Function Definition:

```
Void Route (int N, int i, int j, int c)
{  int distance = (j − i + N) mod N;
   do
   {  if(distance[c] == 1) then  // distance[c] = bit c in binary representation of distance
      {     right =(i + N + 2^(c-1)) mod N
            Route (N, right, j, c+1)
            break;
      }
      c++;  //examine next bit
   }while (c <= r);
```

```
}
```

During initialization P_i creates its routing table examining the binary representation of the distance from P_i to P_j for $0 \leq j \leq (N-1)$. The binary representation of distance is examined for the rightmost 1-bit, *c,* and *config (N, c)* is chosen as the required configuration for sending the message.

Definition of Tree Communication

In the tree communication model, messages can be sent to both the left and right neighbors on a ring for a given configuration. In tree communication, a message is routed by using all RIGHT links or by using all LEFT links.

When routing a message from P_i to P_j:

- If (j > i) and ((j – i) <= N/2), use only *RIGHT* links
- If (j > i) and ((j – i) > N/2), use only *LEFT* links
- If (j < i) and ((i - j) > N/2), use only *RIGHT* links
- If (j < i) and ((i - j) <= N/2), use only *LEFT* links

Theorem: Routing a message from P_i to P_j in an *N*-node MultiRing where $N = 2^r$ and $0 \leq i, j \leq (N-1)$ using tree communication model can be accomplished in *O (log N)* steps.

Proof: Routing using tree communication involves changing each 1-bit of the distance to a 0-bit. When all bits are 0 the message will have reached its destination, P_j.

Base Case: If $P_i = P_j$, distance = 0 and all bits are 0. Thus, the message has reached its destination.

General Case: If $P_i \neq P_j$, distance $\neq 0$ and there is at least 1 bit set to 1.

Let *c* refer to the rightmost 1-bit, where $1 \leq c \leq r$. If all RIGHT links are selected, the message is transmitted to its right neighbor on a ring. As proven with pipeline communication, forwarding a message to a right neighbor on *config (N, c)* reduces the remaining distance by 2^{c-1} for $1 \leq c \leq r$. As a result bit *c* is set to 0. If all LEFT links are selected, the message is transmitted to its left neighbor on a ring. Forwarding a message to a left neighbor on *config (N, c)* adds 2^{c-1} to the remaining distance. As a result bit *c* is set to 0. Bits 1 –(c-1) remain unchanged. Only bits (c+1) – r may be affected by the addition.

$P_{Left} = (i + N - 2^{c-1}) \bmod N$ (by definition of MultiRing)

Distance from P_i to P_j:

$Distance = (j - i + N) \bmod N$ (by definition of distance)

Distance from P_{Left} to P_j in terms of i

$$\begin{aligned} Distance &= (j - ((i + N - 2^{c-1}) \bmod N) + N) \bmod N \\ &= (j - (i + N - 2^{c-1}) + N) \bmod N \\ &= (j - i - N + 2^{c-1} + N) \bmod N \\ &= (j - i + N) \bmod N + 2^{c-1} \\ &= ((\text{Distance from Pi to Pj}) + 2^{c-1} \end{aligned}$$

P_{Left} calculates the remaining distance; now the distance bits numbered from 1 to *c* are all 0. P_{Left} examines the remaining bits (from *c+1* to *r*) for the rightmost 1-bit. At the rightmost 1-bit, P_{Left} will forward the message and the rightmost bit will change to 0. This process continues until bit *r* has been examined. Thus the message will reach P_j in *log N* bits.

Call Statement:

```
If ((j>i)and((j-i)<=N/2)) OR If (j<i)and((i-j)> N/2)
      LINKTYPE = RIGHT_LINK
Else
      LINKTYPE = LEFT_LINK
```

```
      Route (N, i, j, 1, LINKTYPE);
```

Function Definition:

```
Void Route (int N, int i, int j, int c, int LINKTYPE)
{  int distance = (j – i + N) mod N;
   do
   {  If (distance[c] == 1) then   // distance[c] = bit c in binary representation of distance
      {     If (LINKTYPE == RIGHT_LINK)
            {     Right = (i + N + 2^(c-1)) mod N
                  Route (N, right, j, c+1, LINKTYPE)
            }
            Else
            {     Left = (i + N - 2^(c-1)) mod N
                  Route (N, left, j, c+1, LINKTYPE)
.           }

            Break;
      }
      c++;  //examine next bit
   }while (c <= r);
}
```

During initialization P_i creates its routing table examining the binary representation of the distance from P_i to P_j for $0 \leq j \leq (N-1)$. The binary representation of distance is examined for the rightmost 1-bit, *c,* and *config (N, c)* is chosen as the required configuration for sending the message. This configuration is used in conjunction with the link direction to determine the next node.

Definition of Cube Communication

In the cube communication model, each ring configuration establishes links between pairs of nodes that are connected in a single dimension of an N-node hypercube. A node can send data to only one neighbor, but, depending on the ring configuration this may be either the left or right neighbor. A node is connected to its neighbor on *config (N, c)* which id differs by bit *c* in the binary representation of the node ids. If node P_x has a 1 in bit *c* of its node id, it uses the LEFT link to send data to its neighbor on *config (N, c)* for $1 \leq c \leq r$. Otherwise, P_x uses the RIGHT link to send data to its neighbor.

Theorem: Routing a message from P_i to P_j in an *N*-node MultiRing where $N = 2^r$ and $0 \leq i, j \leq (N-1)$ using cube communication model can be accomplished in $O(\log N)$ steps.

Proof: Routing using cube communication involves changing each 1-bit of the distance to a 0-bit. When all bits are 0 the message will have reached its destination, P_j.

Base Case: If $P_i = P_j$, distance = 0 and all bits are 0. Thus, the message has reached its destination.

General Case: If $P_i \neq P_j$, distance $\neq 0$ and there is at least 1 bit set to 1.

Let *c* refer to the rightmost 1-bit, where $1 \leq c \leq r$. If P_i has a 0 in bit *c* on *config (N, c),* the RIGHT link is selected, and the message is transmitted to its right neighbor on a ring. As proven with pipeline communication, forwarding a message to a right neighbor on *config (N, c)* reduces the remaining distance by 2^{c-1} for $1 \leq c \leq r$. As a result bit *c* is set to 0 and bits 1-(c-1) remain unchanged.

If P_i has a 1 in bit *c* on *config (N, c),* the LEFT link is selected, and the message is transmitted to its left neighbor on a ring. As proven with tree communication, forwarding a message to a left neighbor on *config (N, c)* adds 2^{c-1} to the remaining distance. As a result bit *c* is set to 0 and bits 1 –(c-1) remain unchanged. Only bits (c+1) – r may be affected by the addition.

P_{next} is selected as either P_{Left} or P_{Right} depending on the c^{th} bit of its binary address. P_{next} calculates the remaining distance; now the distance bits numbered from 1 to *c* are all 0. P_{next} examines the remaining bits (from $c+1$ to *r*) for the rightmost 1-bit. At the rightmost 1-bit, P_{Next} will forward the message and the rightmost bit will change to 0. This process continues until bit *r* has been examined. Thus the message will reach P_j in $\log N$ bits.

Call Statement:

```
Route (N, i, j, 1);
```

Function Definition:

```
Void Route (int N, int i, int j, int c)
{  int distance = (j – i + N) mod N;
   do
   {  If (distance[c] == 1) then   // distance[c] = bit c in binary representation of distance
      {     If ((i[c] == 1) then    // i[c] = bit c in binary representation of i
                  LINKTYPE = LEFT_LINK
            Else
                  LINKTYPE = RIGHT_LINK
            If (LINKTYPE == RIGHT_LINK)
            {     Right = (i + N + 2^(c-1)) mod N
                  Route (N, right, j, c+1)
            }
            Else
            {     Left = (i + N - 2^(c-1)) mod N
                  Route (N, left, j, c+1)
            }

            Break;
      }
      c++;  //examine next bit
   }while (c <= r);
}
```

During initialization P_i creates its routing table examining the binary representation of the distance from P_i to P_j for $0 \leq j \leq (N-1)$. The binary representation of distance is examined for the rightmost 1-bit, *c,* and *config (N, c)* is chosen as the required configuration for sending the message. This configuration is used in conjunction with the c^{th} bit of the Pi's address to determine the next node.

6.2 Arbitrary Send

All of the algorithms in this chapter assume an automatic switch that repeatedly cycles through all of the configurations. P_i sends a message to an arbitrary P_j by doing the following:

1. It looks in the routing table at the destination address to find the appropriate configuration to send the message.

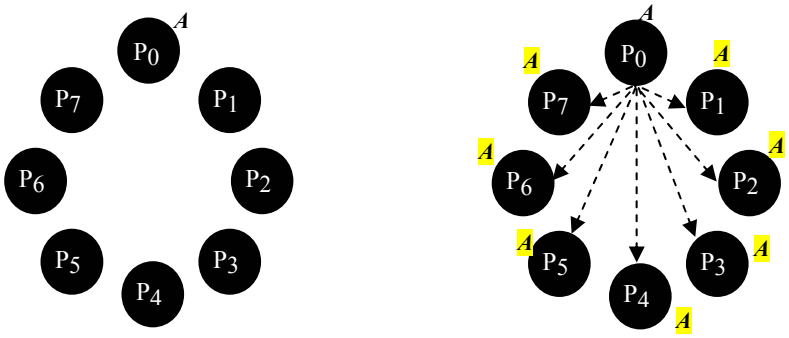
2. It determines the direction (left link or right link) according to the type of communication model used.
3. It waits for the switch to signal that the required configuration exists.
4. It sends the message.

Send (P_{source}, P_{dest}, message);

When a node P_i receives a message, if the destination address matches the node's ID, it keeps the message. However, if the destination address does not match the node's ID, P_i references its local routing table to find the next ring configuration required to forward the message. It will forward the message to the next node in the ring when the required configuration is established.

6.3 Ring Broadcast

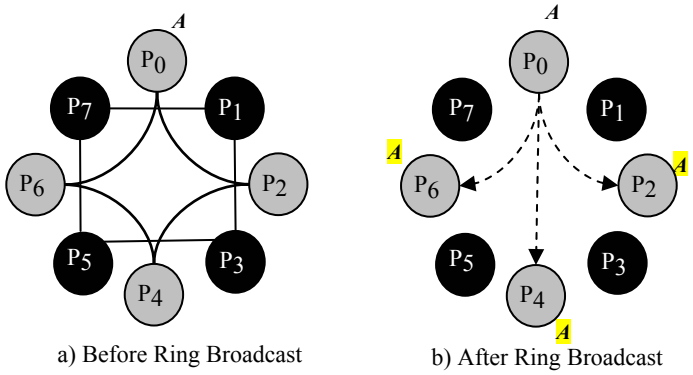
In *broadcasting*, one node wants to send the same message to all of the nodes in the network. In Figure 6.6, P_0 has a message, '*A*' to send to all nodes in the network. After P_0 broadcasts the message, all nodes have a copy of the message. The dashed lines represent the logical distribution of the message.



a) Before broadcast b) After broadcast

Figure 6.6: Broadcasting

Since multiprogramming is possible, the MultiRing may be arranged into several independent rings using a particular configuration; this may be referred to as *config (N, ringConfig)* where $1 \leq ringConfig \leq r$. Ring broadcasting is a special form of broadcasting where a node shares its data only with the other nodes on its ring. In Figure 6.7, an 8-node MultiRing has been divided into two rings of 4 nodes. Node P_0 wants to send its message to the other nodes on its ring. After ring broadcasting, all of the nodes in P_0's ring have a copy of the message. The dashed lines represent the logical distribution of the message.



a) Before Ring Broadcast b) After Ring Broadcast

Figure 6.7: Ring broadcasting

There are two versions of ring broadcasting described in this section: 1) sending individual messages addressed specifically to each node and 2) sending special 'ring broadcast' messages.

6.3.1 Sending Individual Messages

If the original broadcasting node produces an individual message for each node on the ring, time is taken to send copies of the same message to each with a different address. The arbitrary send function, `Send()` is used to send a single copy of the message. For example, if P_0 broadcasts a message to all other nodes in an 8-node MultiRing, it sends seven copies of the message -- one for each node (P_1, P_2, P_3, P_4, P_5, P_6, and P_7).

Messages from P_0:

`Send (P0, P1, message)` -- on *config(8, 1)*

`Send (P0, P2, message)` -- on *config(8, 2)*

`Send (P0, P3, message)` -- on *config(8, 1)*

`Send (P0, P4, message)` -- on *config(8, 3)*

`Send (P0, P5, message)` -- on *config(8, 1)*

`Send (P0, P6, message)` -- on *config(8, 2)*

`Send (P0, P7, message)` -- on *config(8, 1)*

Messages must be transmitted from the broadcasting node in the order in which they were created. This means P_0 cannot send P_2's message before sending P_1's message. Therefore, P_0 sends the messages in consecutive order starting with its right neighbor. Some messages may have to be forwarded through intermediate nodes to reach their final destinations.

Table 6.2 shows the sequence when messages are received if P_0 broadcasts in an 8-node MultiRing and the automatic switch repeatedly reconfigures the MultiRing in consecutive stages from *config(8,1)* to *config(8,3)*. Each entry in the table denotes a message that has arrived at a node. The notation, P0→P1, represents a message destined for P_1 that was sent from P_0. The notation, P01→P5, represents a message destined for P_5 that was sent from P_0 and transmitted through P_1 before reaching the current node. The messages in bold type are those that have reached their destinations. A shaded line indicates a configuration of the network that was not used to transmit data but was a part of the cycle of the automatic switch.

In step 1 on *config(8,1)*, P_1 receives its message from P_0. In step 2 on *config(8,2)*, P_2 receives its message from P_0. In step 4, P_1 receives a message addressed to P_3. In step 5, P_3 receives its message from P_0 that was transmitted through P_1. Looking at the table, it is clear that most messages are not being sent from different nodes in parallel since nodes must process messages in the order in which they arrive. P_1 receives messages for P_3, P_5 and P_7; however, P_3's message must be transmitted before P_5's and P_5's message must be transmitted before P_7's. Only in step 9 do two nodes, P_5 and P_6 receive their message on the same configuration.

Table 6.2: Broadcasting by sending individual messages.

Step	**Configuration Number**	**Receiving Nodes**							
		P_0	**P_1**	**P_2**	**P_3**	**P_4**	**P_5**	**P_6**	**P_7**
1	1		**P0→P1**						
2	2			**P0→P2**					
3	3								
4	1		P0→P3						

5	2				**P01→P3**				
6	3					**P0→P4**			
7	1		P0→P5						
8	2			P0→P6					
9	3						**P01→P5**	**P02→P6**	
10	1		P0→P7						
11	2				P01→P7				
12	3								**P013→P7**

When broadcasting is accomplished by sending individual messages for each node, the original broadcasting node has the tedious work of creating and sending copies of its message. Since messages are processed in consecutive order, the parallelization of transmitting the messages over the network is limited.

If an automatic reconfigurable switch is used to cycle through all available configurations, on some configurations no messages may be transmitted. For example in Table 6.2, in step 2, *config(8,2)* is needed to transmit P_2's message and in step 4 *config(8,1)* is needed to transmit P_3's message. With an automatic switch, after *config(8,2)*, the network will be reconfigured into *config(8,3)* before being reconfigured into *config(8,1)*. Thus in step 3, the network is configured into *config(8,3);* however no nodes use this configuration. The nodes will simply wait until the switch cycles to the next configuration before sending a message.

6.3.2 Sending Broadcast Message

In this version of broadcasting, the workload of copying and messages is distributed among all nodes on the ring. The original broadcasting node sends messages that signal a " ring broadcast." To signify a 'ring broadcast,' the destination address is given a special address that means broadcast, P_x, and the value of *ringConfig* is sent in addition to the message. By using a

special broadcast message instead of sending individual messages addressed to each node, all of the messages can be delivered in (*r- ringConfig +1)* configurations if an automatic switch cycles through configurations starting at *config (N, r)* down to *config(N, ringConfig*), and several messages are transmitted in parallel.

The procedure for initiating and receiving ring broadcast messages differs from the type of communication model used for routing messages. The node initiating the message, P_{source}, may send several messages (pipeline and cube communication models) or one single message (tree communication model). The function LinkType(), which is different for each communication model, is used to decide the LINK to send the message.

6.3.2.1 Initiating Broadcast Messages

If either pipeline or cube communication models are used, P_{source} sends a single broadcast message in each configuration from *config(N, r)* down to *config (N, ringConfig*).

```
//BROADCAST_INITIATE_PIPELINE_OR_CUBE()
 //Transmit on all configurations between config(N, r) and config(N, ringConfig)
For(c=r; c>=ringConfig; c--)
{
      OUTPUT_LINK = LinkType(c)              //get link direction for the configuration
      Send(Psource, Px, ringConfig, message);
}
```

For the pipeline communication model, LinkType() always returns a RIGHT_LINK; thus the message is always sent using RIGHT_LINK. In contrast on the cube communication

model, LinkType() may return either LEFT_LINK or RIGHT_LINK depending on which node P_{source} communicates on a configuration.

If the tree communication model is used, P_{source} only sends *one* message. On *config (N, r)* it sends a broadcast message to its only neighbor.

```
//BROADCAST_INITIATE_TREE
//Transmit on config(N, r)
OUTPUT_LINK = RIGHT_LINK;
Send(P_source, P_x, ringConfig, message);
```

6.3.2.2 Receiving a Broadcast Message

Once a node, P_i, receives a broadcast message, it keeps a copy for itself before re-transmitting the broadcast message. In order to retransmit the message on the right configurations, P_i takes note of the current configuration by which the message was sent, *config (N, c)*. P_i doesn't change the source address when retransmitting the message; the source node ID remains P_{source} for Send ().

If either the pipeline or the cube communication model is used, the message is repeatedly transmitted for each configuration between *config(N, c-1)* and *config (N, ringConfig)*.

```
//BROADCAST_FORWARD_PIPELINE_OR_CUBE
//Transmit on all configurations between config(N, c-1) and config(N, ringConfig)
For(c=c-1; c>=ringConfig; c--)
{
      OUTPUT_LINK = LinkType(c)              //get link direction for the configuration
      Send(P_source, P_x, ringConfig, message);
}
```

If the tree communication model is used, P_i re-transmits one message on its LEFT_LINK and one message on its RIGHT_LINK on *config(N, c-1)*.

```
//BROADCAST_FORWARD_TREE
//Transmit on config(N, c-1)
if (c > ringConfig)
{     c = c-1;
      OUTPUT_LINK = LEFT_LINK
      Send(Psource, Px, ringConfig, message);
      OUTPUT_LINK = RIGHT_LINK
      Send(Psource, Px, ringConfig, message);
}
```

Table 6.3 and Table 6.4 show the sequence of sending 'broadcast' messages when P_0 initiates the broadcast message to all nodes in an 8-node MultiRing. Entries in the tables denote when a message arrives at a node. The destination address, 'PX,' represents a broadcast message. The notation, P04→PX, represents a broadcast message that was sent from P_0 and transferred through P_4 before reaching the current node.

Table 6.3 represents that transmission flow using either the pipeline or cube communication model. In step 1, P_4 receives a broadcast message from P_0. In step 2, P_2 receives a broadcast message from P_0 and P_6 receives a broadcast message from P_4. In step 3, P_1 receives a broadcast message from P_0, P_3 receives a broadcast message from P_2, P_5 receives a broadcast message from P_4 and P_7 receives a broadcast message from P_6.

Table 6.4 represents the transmission flow using the tree communication model. In step 1, P_4 receives a broadcast message from P_0. In step 2, P_2 receives a broadcast message from P_4. In

step 3, P_1 receives a broadcast message from P_2, and P_5 and P_7 receive a broadcast message from P_6.

Table 6.3: Broadcasting using pipeline and cube communication models

Step	Configuration Number	Receiving Nodes							
		P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
1	3					P0→PX			
2	2			P0→PX				P04→PX	
3	1		P0→PX		P02→PX		P04→PX		P046→PX

Table 6.4: Broadcasting using tree communication model

Step	Configuration Number	Receiving Nodes							
		P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
1	3					P0→PX			
2	2			P04→PX				P04→PX	
3	1		P042→PX		P042→PX		P046→PX		P046→PX

Regardless of which communication model is used, all of the nodes will receive the broadcast message. However, the flow by which messages are delivered differs. As a further illustration, it is worthwhile considering what happens when P_2 broadcasts a message. Figure 6.8 demonstrates the flow of messages for each communication model. The differences in transmission flow are seen by comparing step 2 of each communication model. In the pipeline model, P_2 sends 'A' to P_4, and P_6 sends 'A' to P_0 using only the RIGHT_LINK. With the cube communication model, P_2 sends 'A' to P_0 and P6 sends 'A' to P_4 using only the LEFT_LINK.

With the tree communication model, P_6 sends 'A' to P_0 using RIGHT_LINK and to P_4 using the LEFT_LINK.

Figure 6.8 also illustrates that as required for effective multiprogramming, messages are sent only to nodes on the ring. The automatic switch would begin at *config (N, r)* and continue until *config(N, rconfig)*. In the 8-node MultiRing when broadcasting on a ring with only two nodes, *ringConfig* =3; therefore, stop broadcasting after step 1. However, when broadcasting on a ring with all 8 nodes, *ringConfig* =1 so stop broadcasting after step 3.

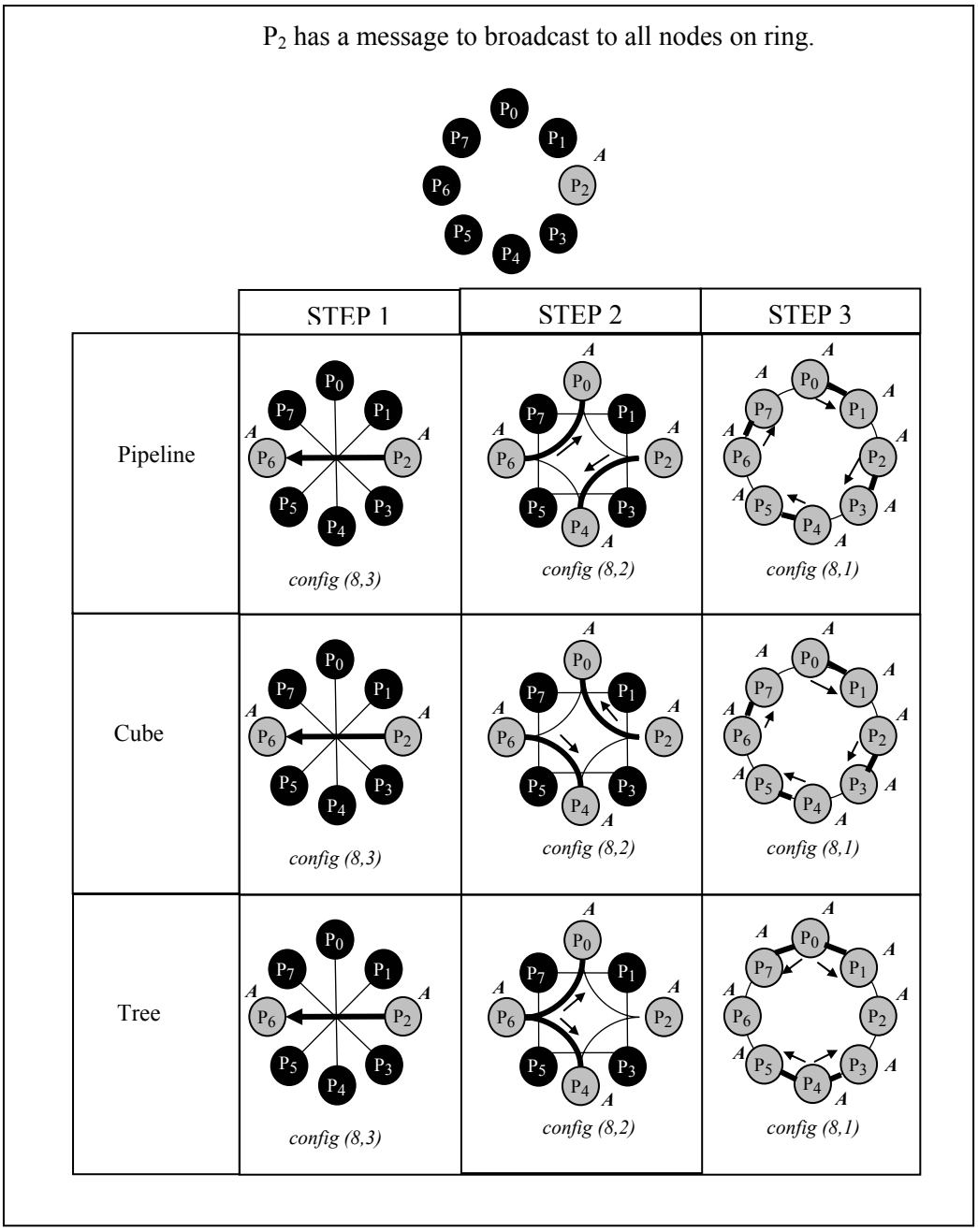


Figure 6.8: P_2 broadcasts a message using different communication models.

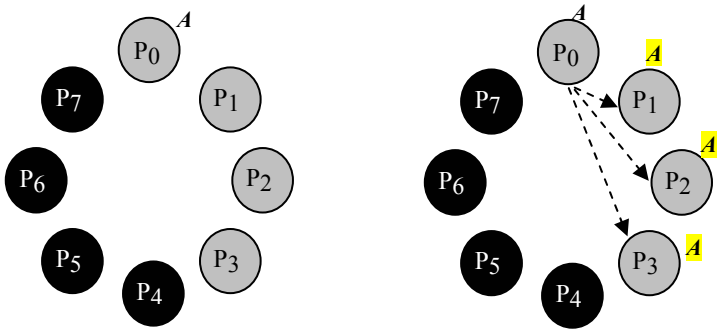
With automatic switch reconfiguration, a node simply waits until the required configuration exists before sending a message. As with the previous version of broadcasting, messages are dispatched in consecutive order. In the best case, only *r* different switch configurations are needed for the messages to be delivered. If however, the time taken for a

node to recognize the message is a 'broadcast' message, and to get it ready to forward on the next configuration is longer than the time the automatic switch waits before reconfiguring, more than one cycle through the different configurations may be necessary.

6.4 Group Broadcasting

Group broadcasting is a special form of broadcasting in which a node needs to share its data with only a portion of the nodes in the network. In group broadcasting, a node sends a message to the other nodes in its *logical* group of consecutive nodes. In a 2^r-node MultiRing, the nodes can be evenly divided into $g = 2^i$ logical groups each with $s=2^{r-i}$ nodes where i is an integer and is $0 \leq i \leq r-1$. The groups are numbered from G_0 to G_{g-1} and the nodes in group G_x are numbered from P_{xs} to $P_{(x+1)s - 1}$.

In Figure 6.9, the 8-node MultiRing has been divided into two logical groups. Each group has a different shade. In Figure 6.9 a) P_0 has a message, 'A', to send to all nodes in its logical group. In Figure 6.9 b) after a group broadcast, all of the nodes in P_0's group have a copy of the message. The dashed lines represent the logical distribution of the message.



a) Before Group Broadcast b) After Group Broadcast

Figure 6.9: Group broadcasting

As illustrated in Figure 6.10, the following possible logical grouping divisions are possible on a 16-node MultiRing: one group of sixteen nodes, two groups of eight nodes, four groups of four nodes and eight groups of two nodes. If the network has been divided into four groups of four nodes (g=4 and s=4 as in Figure 6.10c), the groups are numbered G_0, G_1, G_2 and G_3. The nodes in G_0, are P_0, P_1, P_2 and P_3. The nodes in G_1 are P_4, P_5, P_6 and P_7. The nodes in G_2 are P_8, P_9, P_{10} and P_{11}. The nodes in G_3 are P_{12}, P_{13}, P_{14} and P_{15}. The nodes in each group are consecutively numbered.

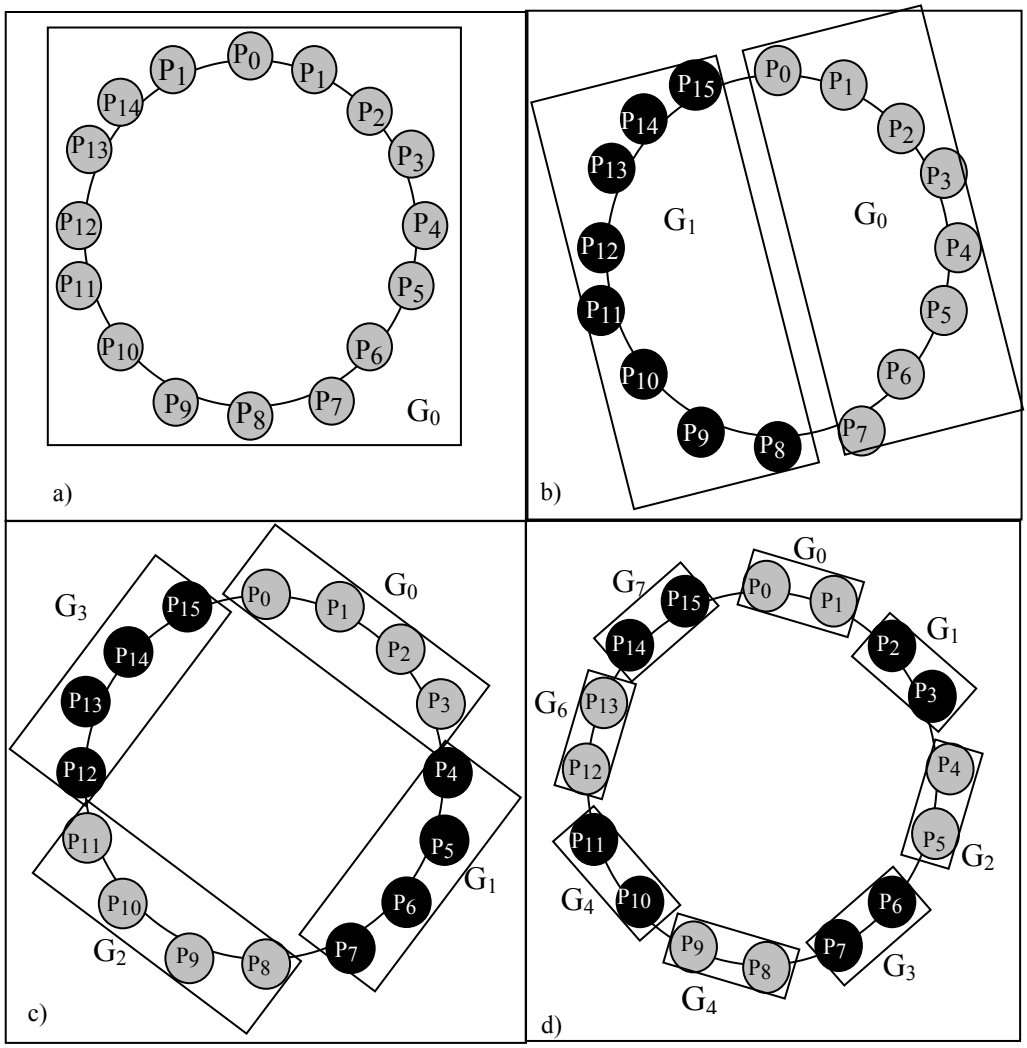


Figure 6.10: Logical grouping divisions of 16 nodes on a 16-node MultiRing.

If the MultiRing is arranged into several independent rings using a particular ring configuration, *config (N, ringConfig)* where $1 \leq ringConfig \leq r$, logical groups can be formed with only the nodes on *ringConfig*. For example, if a 16 node MultiRing is configured into 2 rings of 8 nodes, *ringConfig* =2 and each ring can be logically grouped into one group of 8 nodes, 2 groups of 4 nodes and 4 groups of 2 nodes. Figure 6.11 shows the logical groups of one ring of 8 nodes on a 16-node MultiRing.

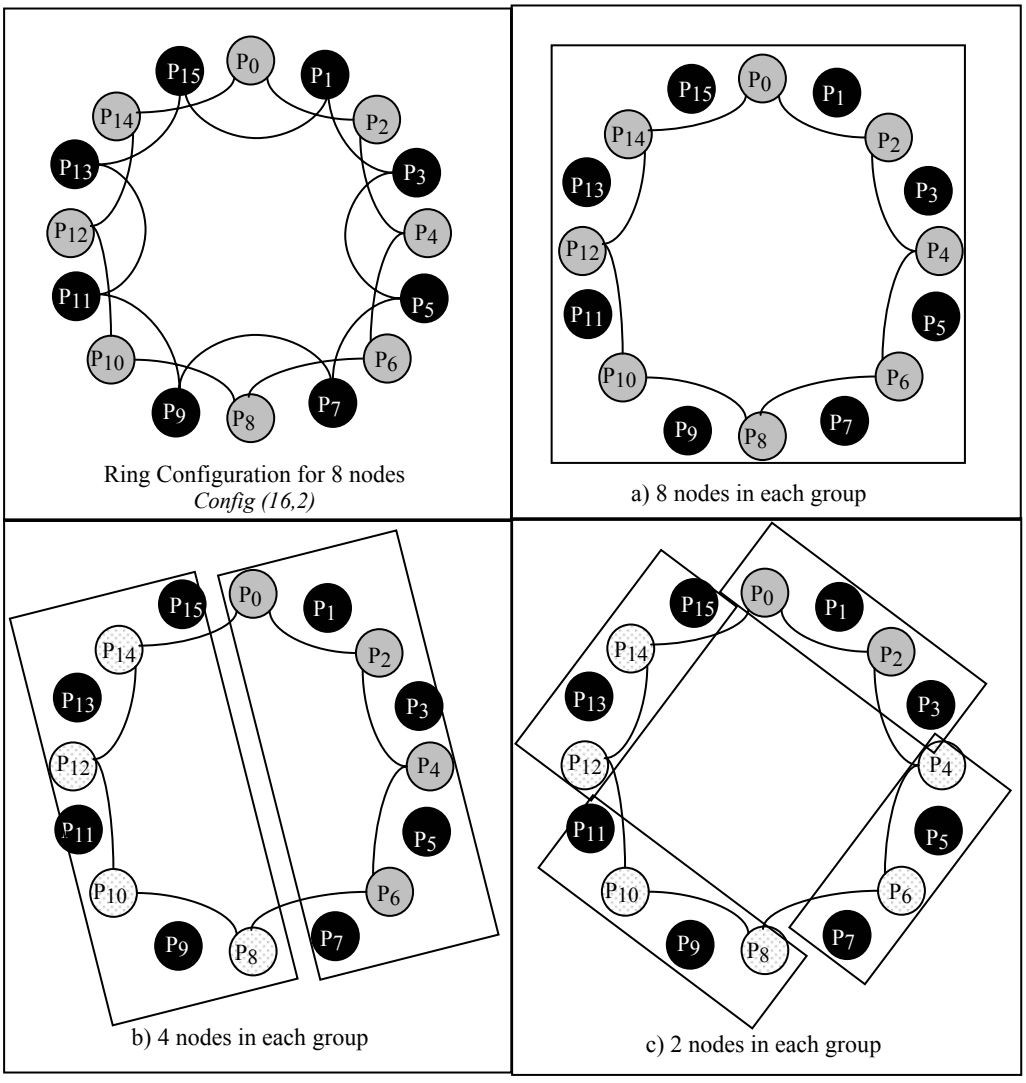


Figure 6.11: Logical grouping divisions of 8 nodes on a 16-node MultiRing.

Individual messages can be sent to each group member; however, a special 'group broadcast' flag can also be used. If the network has been divided into 2^i logical groups to broadcast to the other nodes in the logical group, a node sends a 'group broadcast' message on each configuration starting with *config(N, groupConfig)* and continuing to the last required configuration, *Config(N, ringConfig)* where *groupConfig = r-i*. Each node receiving a 'group broadcast' message takes note of the configuration number, *config(N, c)*, which was used to send the message and the last required configuration number, *groupConfig,* and forwards the message once for each configuration starting at *config(N, c-1)* and ending with *config(N, ringConfig)*.

6.4.1 Initiating a 'Group Broadcast' Message

Using the pipeline communication model, in order to limit the number of nodes outside the group that transmit a message, all nodes initially send the group broadcast message to the lowest numbered node, P_{low}, in the logical group and only let Plow 'group broadcast' the message to the other members of its group. An arbitrary node, P_i, can broadcast to a group by:

1) finding the address of P_{low} for its group,

2) using an Arbitrary Send to send the message to P_{low} and

3) having P_{low} group broadcast the message.

A node, P_i determines the P_{low} of its group with the formula: $P_{low} = (s\ (P_i \text{ div } s))$, where *s* is the number of nodes in a group and *div* returns an integer quotient. For example, if there are two groups of four nodes in an 8-node MultiRing, the lowest node in P_3's group is P_0 (i.e. 4 * (3 div 4) = 0).

Using the pipeline or cube communication model on a network that has been divided into 2^i logical groups, a node, $P_{source,}$ that has a message it wants to group broadcast, forwards the

message *r-i* times—once for each configuration from *config(N, groupConfig)* down to *config(N, ringConfig)* .

```
//GROUP_BROADCAST_INITIATE_PIPELINE_OR_CUBE
groupConfig = r-i; // first configuration needed to send data if there are 2^i groups is r-i.
For (c=groupConfig; c>=ringConfig; c--)
 {    OUTPUT_LINK = LinkType(c) //get link direction for the configuration
      Send(P_source, P_group, message);
}
```

If the tree communication model is used, P_{source} sends messages to both neighbors on *config (N, groupConfig)*.

```
//BROADCAST_INITIATE_TREE
//Transmit on config(N, groupConfig)
OUTPUT_LINK = RIGHT_LINK;
Send(P_source, P_group,message);
If (groupConfig != r)  //more than one neighbor
{     OUTPUT_LINK = LEFT_LINK;
      Send(P_source, P_group, message);
}
```

6.4.2 Receiving a 'Group Broadcast' Message

When a node receives a 'group broadcast' message, it takes note of the configuration on which it received the message, *config(N, c).* It forwards the message for each configuration from *config(N, c-1)* down to *config(N, ringConfig)*.

```
//GROUP_BROADCAST_FORWARD_PIPELINE_OR_CUBE
For (c=c-1; c>= ringConfig; c--) //for all remaining configurations
{     OUTPUT_LINK = LinkType(c);        //get link direction for configuration
      Send(P_source, P_group, message);
}
```

Using the tree model and the group broadcasting flag, a node will send to both neighbors, but sometimes the neighbor is not a part of the logical group. A node will only keep the message if it is a part of the group, but it still forwards the message to both neighbors in the next configuration until *config(N, ringConfig)* is reached.

```
//BROADCAST_FORWARD_TREE
//Transmit if not on config(N, ringConfig)
if (c != ringConfig)
{   c--;
   OUTPUT_LINK = RIGHT_LINK;
   Send(P_source, P_group, message);
   OUTPUT_LINK = LEFT_LINK;
   Send(P_source, P_group, message);
}
```

With the cube communication model, only the nodes in the logical group are involved in message transmission of group broadcasts. For example, in an 8-node MultiRing divided into two logical groups, if a node in G_0 wants to 'group broadcast' a message, using *config(8,1) and config(8,2)*, it will only affect nodes in group G_0. If the pipeline or tree models were used some messages will have to be transferred to other groups before returning to G_0. In order for P_3 to send data to P_0 on a pipeline model, the message must be transferred to P_4 before reaching P_0.

In Figure 6.12, an 8-node MultiRing is divided into two logical groups. In order for P_3 to send a message to all nodes in the group, it first sends the message to P_0 and P_0 group broadcasts the message.

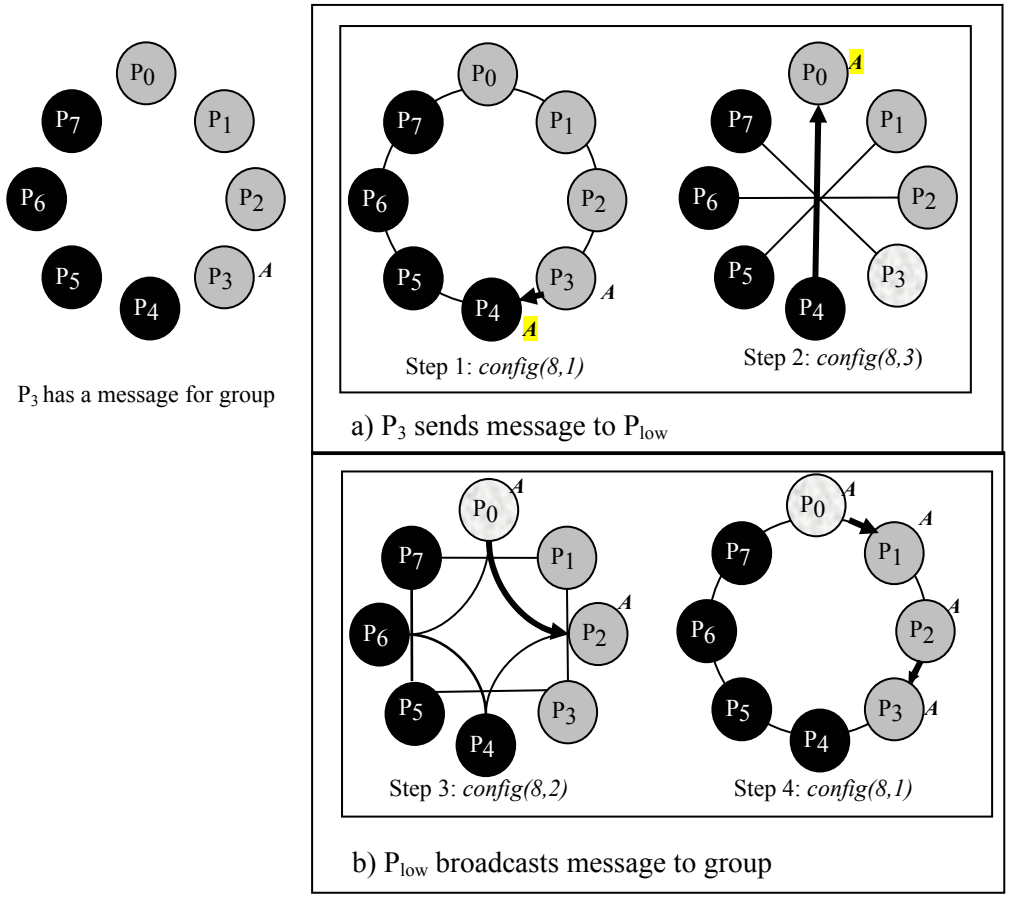


Figure 6.12: Group broadcasting using pipeline communication on a MultiRing.

Any node in a cube MultiRing can send a 'group broadcast' message. In Figure 6.13, the nodes on an 8-node cube MultiRing are divided into two logical groups, and P_3 'group broadcast's a message to the nodes in its group without having to send the message to P_0, the lowest numbered node in its group.

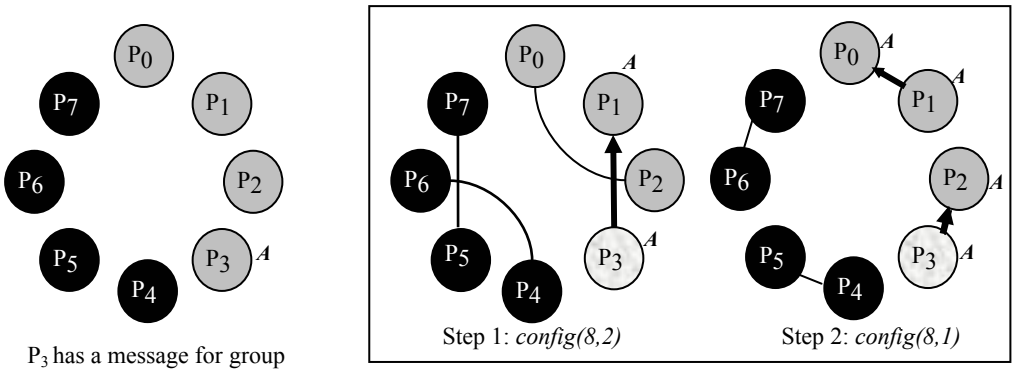


Figure 6.13: Group broadcasting using cube communication on a MultiRing

A lot of unused messages may be sent over the network if the tree communication model is used. In Figure 6.14, P_0 and P_3 group broadcast messages using the tree model on an 8-node MultiRing that has been divided into two logical groups. In Figure 6.14 a) when P_3 is the original broadcaster, P_5 receives an unused message in *config(8,2)* and in *config (8,1)* nodes P_4 and P_6 receive unused messages. In Figure 6.14 b) when P_1 is the original broadcaster, P_7 receives an unused message in *config(8,2),* and P_6 and P_3 receive unused messages in *config (8,1)*.

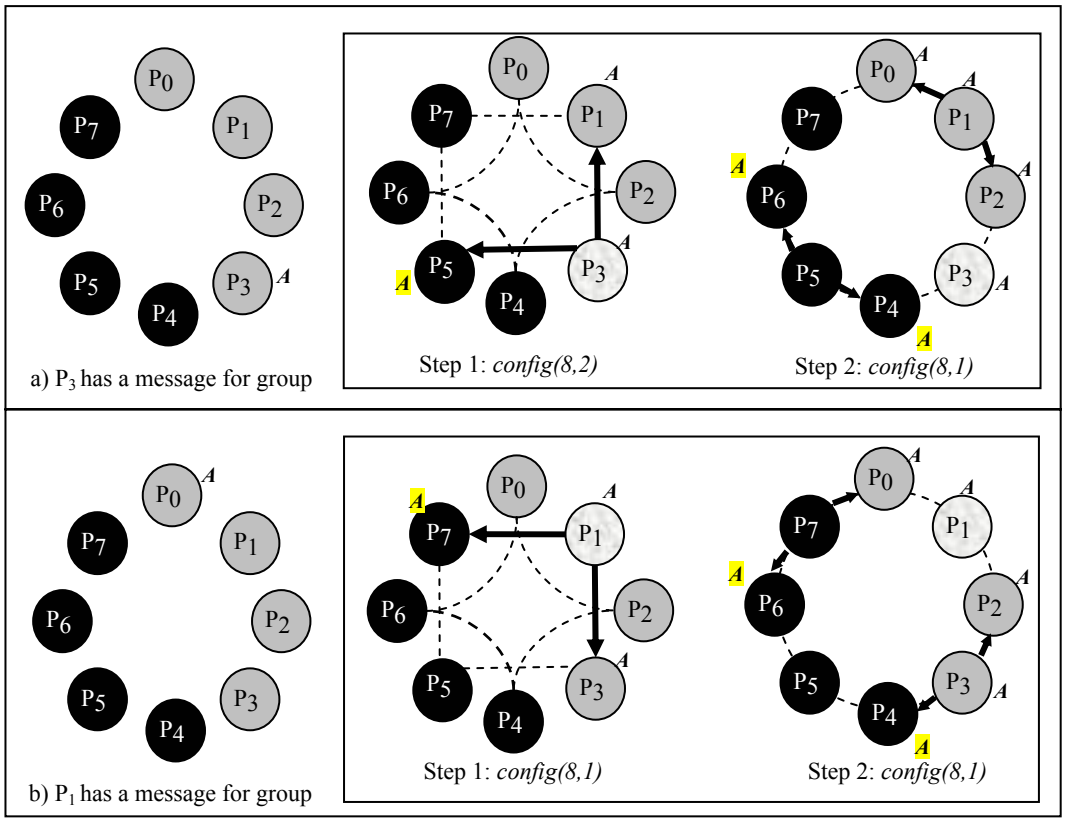


Figure 6.14: Group broadcasting using tree communication on a MultiRing

6.5 Distributing

In *distributing* one node has a large message that contains a different tile (section) for each node on its ring in *config(N, ringConfig)* for $1 \leq ringConfig \leq r$. This node wants to distribute the tiles to all of the nodes on its ring, where each node retains only one tile of the original message. In Figure 6.15, P_0 has a message with 8 tiles to distribute with all of the nodes on the ring of 8 nodes, where *ringConfig* =1. The tiles are labeled '0', '1', '2', '3', '4', '5', '6', and '7'. After completing the distributing message-passing operation, each node contains one tile of the message, and the tiles are arranged in order from P_0 to P_{N-1}.

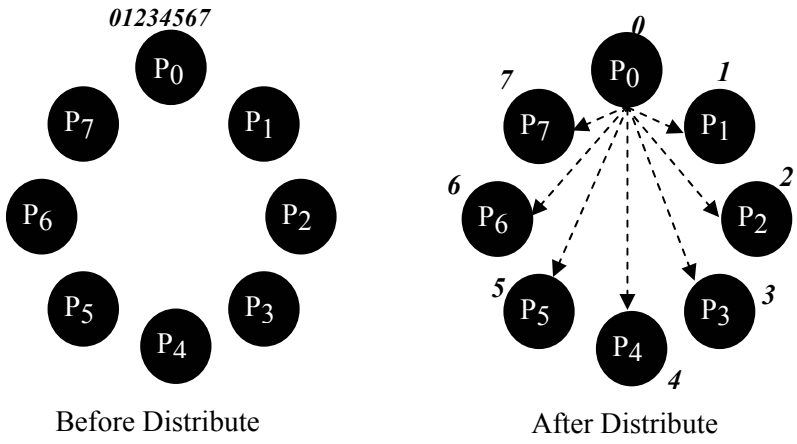


Figure 6.15: Distributing

Although it is possible to send individual messages containing a single tile to each node, this section focuses on sending a special 'distribute' message. To signify a 'distribute' message, the destination address is given a special address that means distribute, P_y, and the value of *ringConfig* is sent in addition to the message. The transmission flow of messages as they are distributed differ depending on the communication model used. Following pipeline and cube communication models, for each configuration, the following occurs: a node splits its message in half, retains half and sends half of the message to the next node. However, pipeline and cube communication models differ in deciding whether the first or last half of the message sent to the neighboring nodes. With the tree communication model, a node extracts its one tile from the middle of the message and sends left half of the message to the left neighbor and the right half of the message to the right neighbor. All of the algorithms require the switch to reconfigure in reverse order, starting with *config(N, r)* down to *config(N, ringConfig)*.

6.5.1 Initiating a Distribute Message

Using a pipeline communication model, the original distributing node, P_{source}, sends a distribute message in each configuration from *config (N, r)* down to *config(N, ringConfig)*. If

order is important in the distribution of the message, P_{source} must arrange the message in consecutive order starting with the tile for P_{source}, concatenating the tile for P_{source}'s right neighbor on *config(N, ringConfig)* and continuing adding tiles in consecutive order around the ring until the last tile for P_{source}'s left neighbor on *config(N, ringConfig)* is added. For example, if P_0 is initiating a distribute message to all tiles in the network, the tiles are arranged '`01234567.`' However if P_2 is initiating the distribute message, all tiles are arranged in the order '`23456701.`'

P_{source} sends a distribute message in each configuration however, the size of the message is reduced by half after each configuration. Starting on *config (N, r*), P_{source} splits the message in half, sends the right half and keeps the left half. The left half is now considered the complete message for the next configuration. P_{source} recursively divides its data into half, keeping the first half and sending the last half until in the last configuration, *config(N, ringConfig)* it keeps a single individual tile and sends a single individual tile.

```
//DISTRIBUTE_INITIATE_PIPELINE
//Organize the data, if necessary, in a consecutive order around ring ending with left neighbor
Organize(message);

//Transmit on all configurations between config(N, r) and config(N, ringConfig)
For (c = r; c >= ringConfig; c--)
{     Half (message, First, Last); //Divide the message into half
      OUTPUT_LINK = RIGHT_LINK;   //the right link is always used for unidirectional
      Send (Psource, Py, ringConfig, Last); //Send the last half of message
      message = First;
}
```

Using the cube communication model to distribute the tiles, there is no need to organize the message in consecutive order on the ring. The order '01234567' will be preserved no matter which node initiates the broadcast. The message is repeatedly split into two equal parts, and half is sent to the neighboring node for each configuration from *config(N, r)* down to *config (N, ringConfig)*. If P_{source} uses its left link on a configuration, it sends the *first* half of the message and keeps the last half. Alternatively, if P_{source} uses its right link on a configuration, it sends the *last* half and keeps the first half.

```
//DISTRIBUTE_INITIATE_CUBE
//Transmit on all configurations between config(N, r) and config(N, ringConfig)
For (c = r; c >= ringConfig; c--)
{     Half (message, First, Last); //Divide the message into half
      OUTPUT_LINK = LinkType(c) //get link direction for the configuration

      If (OUTPUT_LINK == RIGHT_LINK)
      {     Send (Psource, Py, ringConfig, Last); //Send the last half of message
            message = First;
      }
      Else
      {     Send (Psource, Py, ringConfig, First); //Send the first half of message
            message = Last;
      }
}
```

When the tree communication model is used to distribute tiles, the tiles are first arranged in consecutive order around the ring as described in the pipeline model. The first tile is extracted

from the message; P_{source} keeps this single tile and sends the remainder of the message to its neighbor in *config(N, r)*.

```
//DISTRIBUTE_INITIATE_TREE
//Organize the data, if necessary, in a consecutive order around ring ending with left  neighbor
Organize(message);
//Extract First Tile and keep it
ExtractFirstTile(message, tileA, remainingTiles);
message = tileA;
//Transmit one message on config(N, r)
OUTPUT_LINK = RIGHT_LINK;
Send (Psource, Py, ringConfig, RemainingTiles); //Send all remaining tiles
```

6.5.2 Receiving a Distribute Message

A node, P_i, that receives a 'distribute' message keeps track of the configuration, *config(N, c)* and only transmits a portion of the message on following configurations. The source node ID for Send() doesn't change when forwarding the modified message—that is, P_{source} remains as the source ID.

If the pipeline communication model is used, the size of the message is reduced by half after each configuration, P_i sends the right half of the message, and it keeps the left half. For each configuration between *config (N, c-1)* and *config(N, ringConfig),*divide the message in half and send the right half to the right neighbor.

```
//DISTRIBUTE_FORWARD_PIPELINE
//Transmit on all configurations between config(N, c-1) and config(N, ringConfig)
For (c = c-1; c >= ringConfig; c--)
{     Half (message, First, Last); //Divide the message into half
      OUTPUT_LINK = RIGHT_LNK;   //the right link is always used for unidirectional
      Send (Psource, Py, ringConfig, Last); //Send the last half of message
      message = First;
}
```

If the cube communication model is used, P_i follows the same procedure that P_{source} used to initiate the distribute message:

```
//DISTRIBUTE_FORWARD_CUBE
//Transmit on all configurations between config(N, c-1) and config(N, ringConfig)
For (c = c-1; c >= ringConfig; c--)
{     Half (message, First, Last); //Divide the message into half
      OUTPUT_LINK = LinkType(c) //get link direction for the configuration

      If (OUTPUT_LINK == RIGHT_LINK)
      {     Send (Psource, Py, ringConfig, Last); //Send the last half of message
            message = First;
      }
      Else
      {     Send (Psource, Py, ringConfig, First); //Send the first half of message
            message = Last;
      }
}
```

If the tree communication model is used, P_i extracts the middle tile as its own and transmits the first half of message before the middle tile on its LEFT_LINK and the right half of message after the middle tile on its RIGHT_LINK on *config(N, c-1)*.

```
//DISTRIBUTE_FORWARD_TREE
//Extract First Tile and separate remaining message into first and last portions
ExtractMiddleTile(message, tileA, first, last);
message = tileA;
//Transmit one message on config(N, c-1)
if (c > ringConfig)
{     c = c-1;
      OUTPUT_LINK = RIGHT_LINK;
      Send (Psource, Py, ringConfig, last); //Send last half of message
      OUTPUT_LINK = LEFT_LINK;
      Send (Psource, Py, ringConfig, first); //Send first half of message
}
```

In Figure 6.16, P_6 distributes a message to all other nodes using the different communication models. The cube communication model requires less setup time when compared to pipeline and tree models, since the message does not have to be reordered depending on which node is distributing it.

The tree communication model requires the initiating distributor to send a very large message in *config(N, r)*. However, an advantage of using the tree communication model is that each node that receives a distribute message sends only two messages regardless of the number of configurations. In comparison with splitting the data in half, extracting the middle tile allows

the node to begin processing its single tile instead of waiting until all nodes have their individual tile.

In the best case, if the automatic switch reconfigures in reverse order from *config(N, r)* down to *config (N, 1),* at most *r* different switch configurations are needed for the messages to be distributed. If however, the time taken for a node to recognize the message is a 'distribute' message and to get it ready to forward on the next configuration is longer than the time the automatic switch waits before reconfiguring, more than one cycle through the different configurations may be necessary.

In the worst case, the automatic switch only reconfigures in sequential order from *config(N, 1)* to *config (N, r).* In this case, a node that receives a 'distribute' message may have to wait for the switch to pass *r-1* configurations before it sends its message.

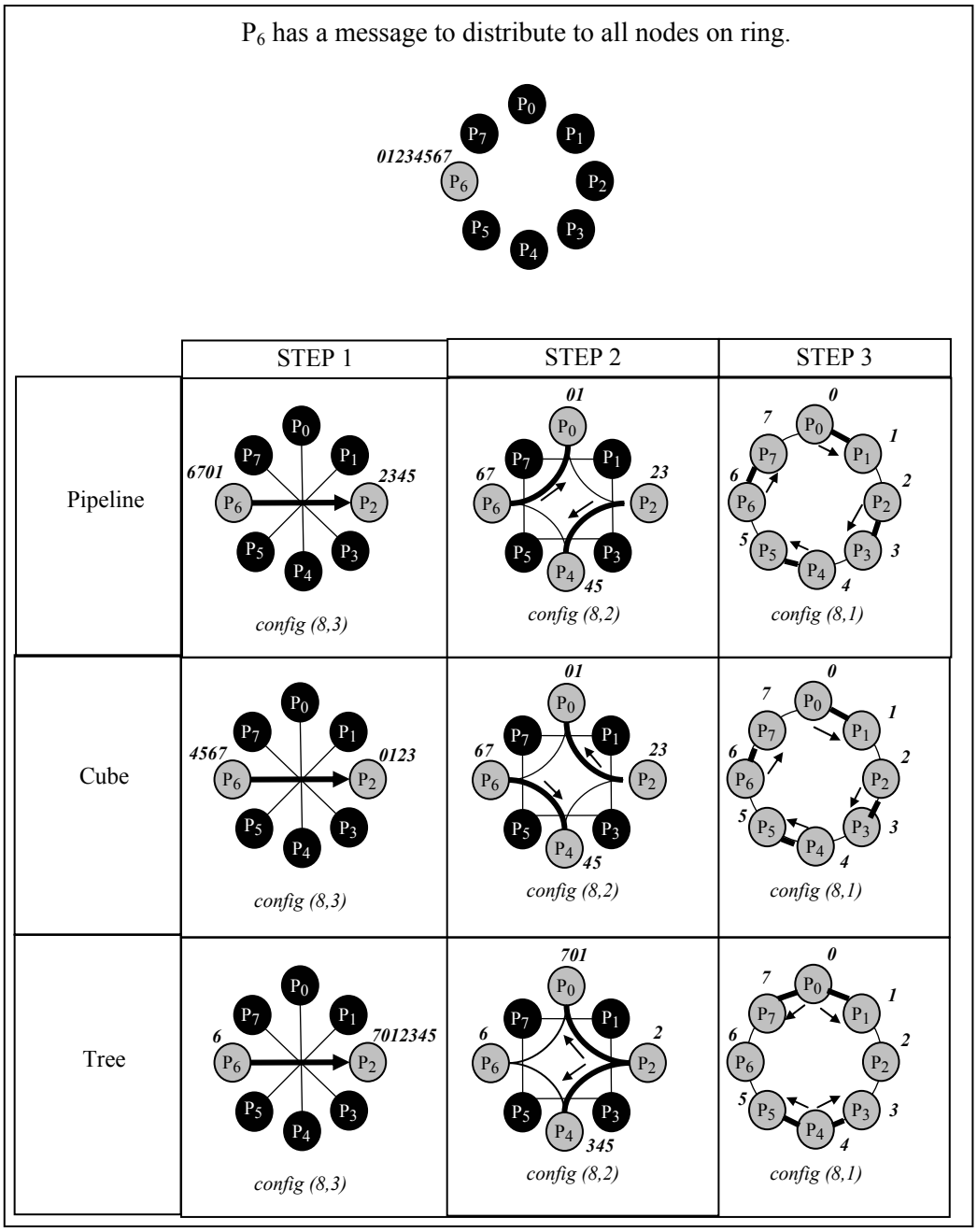


Figure 6.16: P_6 distributes messages using different communication models

6.5.3 Image Tile Broadcasting

Distributing is also called image tile broadcasting when an image is divided into a 2D grid of tiles and P_0 disperses the tiles among the nodes arranged in a 2-D grid. In Figure 6.17, an

image is divided into 16 tiles, 'A'-'P.' The 2-D grid of nodes is arranged in so that each has 1 tile.

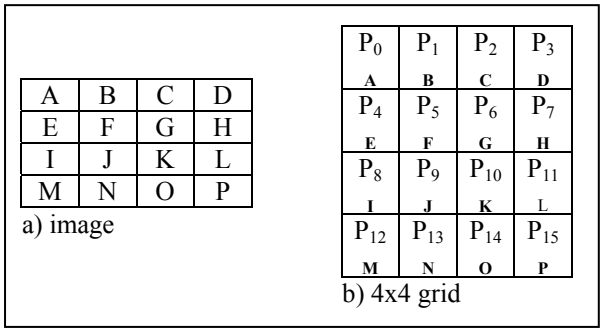


Figure 6.17: Image tiles are distributed to nodes in a 2-D grid.

When the pipeline communication model is used to distribute the image tiles, the tiles are distributed down the 1st column, then across the rows. Figure 6.18 shows the sequence of distribution for the 16 tiles on the 2-D grid.

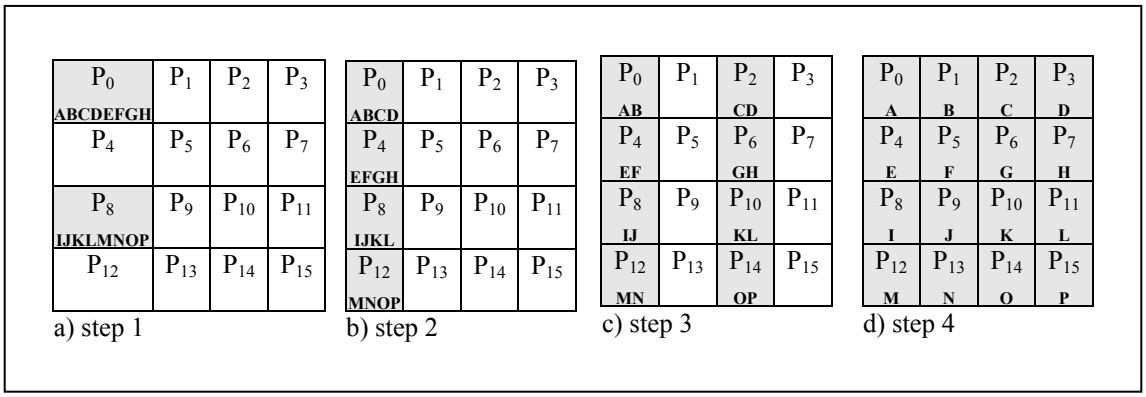


Figure 6.18: Sequence of image tile broadcasting of 16 tiles in a 4x4 grid.

In Chapter 4, it was noted that smaller grids containing less than *N* nodes can be mapped onto the MultiRing. In those cases, a special algorithm is used to distribute the tiles. First one must determine the splitting point, since the tiles may not be divided evenly between nodes at

each distribute phase. The next step is to determine if the right neighbor on the ring in the current configuration is on the grid; only the last half of the tiles are sent if the right neighbor is in the grid. The complete algorithm is given below:

```
IMAGE_TILE_BROADCASTING (R, C, startConfig, colConfig)
// startConfig for initial broadcaster, P0 is r,  otherwise it is c-1 when the message was received on config(N,c)
startConfig is c-1
// colConfig give # of rings in gird

For c= startConfig down to 1
{
    Pj = Pright(c) //calculate Right neighbor on config (N,c)
   (pr, pc) = gridCoordinates (Pj, N, colConfig) //calculate the grid coordinates
    //Determine split point
    if (R != 1)
    { lastR = ⌊R/2⌋
      lastC = C
      firstR = R - lastR
      firstC = C
    }
    Else if (C != 1)
    { lastR = R
      lastC = ⌊C/2⌋
      firstR = R
      firstC = C-lastC
     }
    Else
      firstR = firstC = lastR = lastC= 1;
```

```
  splitPoint = firstR*firstC;


  if (pr < R) && pc < C)   //inside boundary
  {   //split data according to splitpoint
      split(message,splitPoint,first, last)


      Send (Psource, Pj, lastR, lastC, last);


      message = first;
      R = firstR;
      C = firstC;
   }
}
```

In Figure 6.19, an image is divided into 9 tiles in the form of a 3x3 grid. These tiles are distributed down the 1st column, then across the rows.

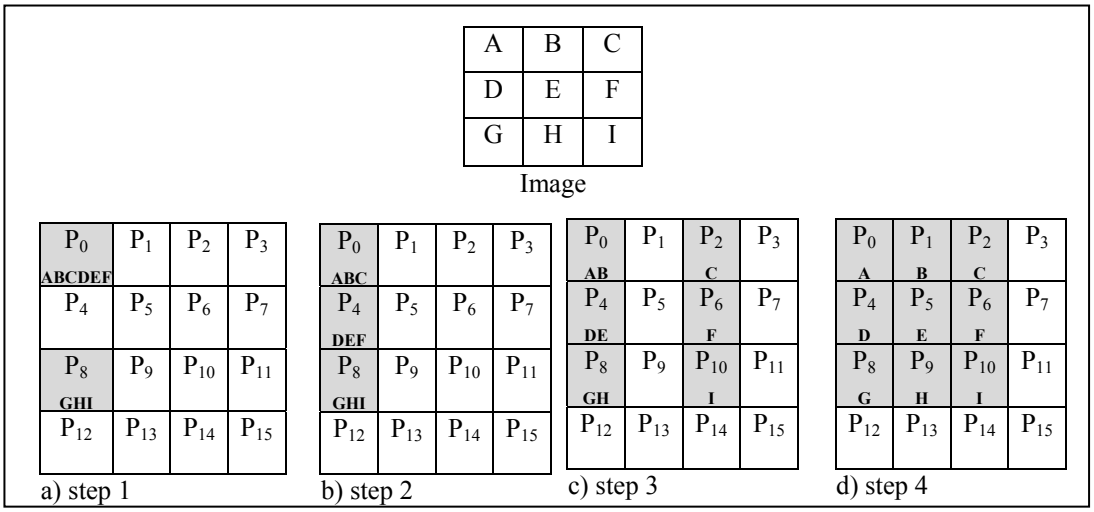


Figure 6.19: Distributing 9 tiles in a 16-node grid.

6.6 Concluding Remarks

This chapter contains special routing and message-passing algorithms for the MultiRing that are specially designed for its reconfigurable nature. Different algorithms were sometimes necessary to be able to transmit messages using pipeline, cube and tree communication models. However, it was interesting to note that the initial configuration required to send a message from one node to an arbitrary node is identical for all communication models.

The four message-passing operations described are arbitrary send, ring broadcasting, group broadcasting and distributing. Arbitrary Send is really the only message-passing operation that is required. However, using special broadcast/distribute messages instead of sending individual messages addressed to each node can be more efficient since several messages can be transmitted in parallel, and all of the messages can be delivered in O(log *N)* different configurations.

Group broadcasting is best implemented using the cube communication since only the nodes in the logical group are involved in broadcasting messages. Also, in practice, the cube communication model required less setup time when distributing individual tiles when compared to pipeline and tree models, since the message did not have to be reordered depending on which node was distributing it. However, an advantage of using the tree communication model is that when a node receives a special message it sends at most two forward-messages regardless of the ring configuration. Also, algorithms for the pipeline communication model are needed if the MultiRing is unidirectional.

A MultiRing can be arranged into several independent rings using a particular configuration referred to as *config (N, ringConfig)* where $1 \leq ringConfig \leq r$. All algorithms

presented that support multiprogramming by restricting communication involve nodes on the main ring configuration only.

With automatic switch reconfiguration, a node simply waits until the required configuration exists before sending a message. In the best case, only r different switch configurations are needed for the messages to be delivered. If, however, the time taken for a node to recognize the message is a special message, and to get it ready to forward on the next configuration is longer than the time the automatic switch waits before reconfiguring, more than one cycle through the different configurations may be necessary.

CHAPTER 7 :MULTIPROGRAMMING ON THE MULTIRING

Multiprogramming occurs when multiple independent applications run together on the same machine. Different applications either "time-share" the machine's processor, where each application is given a time-slice to execute, or "space-share," where applications run on disjoint subsets of the machine's processor, or both [26]. In this dissertation the focus is on the MultiRing, and multiprogramming is defined as multiple programs running together on the same MultiRing network where each program uses a disjoint subset of the MultiRing's nodes. An example of multiprogramming on a 16-node MultiRing is to have 8 nodes executing application *A*, another 4 nodes working on application *B* and the remaining 4 nodes working on an application *C*. The communication strategies and routing operations described in Chapters 5 and 6 have been developed to maintain independent rings that allow parallel exchange of data between neighboring nodes on the same ring.

This chapter shows how to maintain independence between rings on the MultiRing and presents several parallel strategies for exchanging data between neighboring processes for the pipeline, cube and tree communication models. In addition, the chapter shows how enabling multiprogramming makes it possible to add new nodes quickly with limited disruption to the system. Section 1 describes independent ring formation, section 2 presents parallel exchange of data strategies, section 3 describes how to add nodes to the MultiRing and section 4 presents the costs of multiprogramming on the MultiRing. Concluding remarks are provided in section 5.

7.1 Independent Rings

The general layout of the MultiRing is a star format – all nodes connect to a central switch. As the switch changes configurations, different neighboring nodes are connected forming a collection of rings. In *config (N, 1)* a single ring of all nodes exists, and in *config (N, 2)* a collection of rings with only two nodes is formed. Multiprogramming on the MultiRing involves one program using a subset of nodes that forms a single ring in one configuration of the network. The main ring configuration, *config (N, ringConfig),* for a program is selected as the smallest ring that has the number of nodes required. Chapter 5 presented the pipeline, cube, tree and grid communication strategies for selecting the main ring configuration for applications that feature a pipeline, cube, tree or grid algorithm.

To be considered independent rings, the nodes must only communicate with others available on the main ring configuration. To ensure that messages are not routed, even indirectly, through nodes not on *config (N, ringConfig),* only switching configurations from *config (N, ringConfig)* to *config (N, r)* can be used for communication. Configurations, *config (N, 1)* thru *config (N, ringConfig-1)* connect nodes to neighbors that do not exist on the same main configuration.

Chapter 6 presented special routing operations that featured communicating using configurations from *config (N, r)* down to *config (N, ringConfig*). These algorithms differ from previous routing operations [4, 8, 9] in that the configurations change in descending order. Arabnia et al [4, 8, 9] had only considered routing on a unidirectional MultiRing. In this dissertation, simple algorithms have been developed that would be suitable for either unidirectional or bidirectional communication strategies. The pipeline communication strategy

works well on a unidirectional MultiRing; however the tree communication model is created for a bidirectional MultiRing.

It will be possible to develop algorithms for a switch that reconfigures in ascending order from *config (N, ringConfig)* up to *config (N, r);* however, these algorithms will be more complicated. For example, broadcasting using the tree communication model on a descending switch requires each node to broadcast at most two messages, one to its left neighbor and one to its right neighbor. If an ascending switch is used, the original broadcasting node will have to send messages to both neighbors on multiple configurations, and nodes that receive the broadcast message will have to forward the message on multiple configurations as well. In Figure 7.1, P_2 broadcasts on a 8-node network for a switch that cycles in descending order and ascending order.

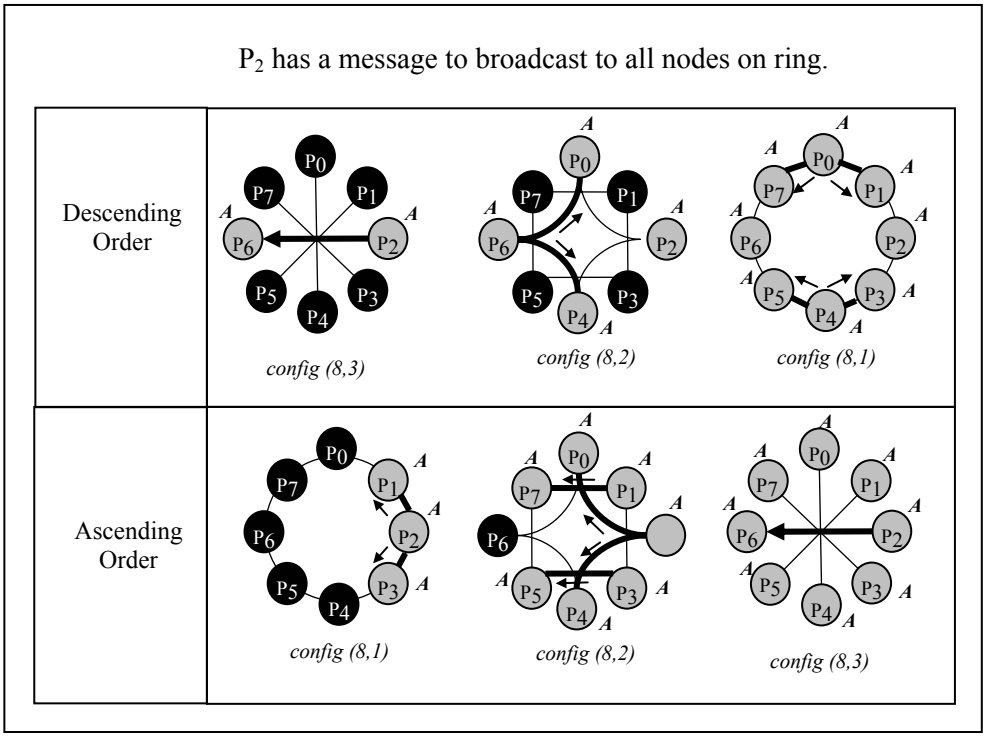


Figure 7.1: P2 broadcasts a message using on descending and ascending switches.

7.2 Parallel Exchange of Data

All parallel algorithms should be designed to prevent deadlock. Deadlocks may occur when two nodes attempt to send to each other simultaneously. There are two ways to send data, blocking send and non-blocking send. With blocking send, a sender node must wait until the receiver node is ready to receive a message. If both nodes are trying to send a message to each other, they are both waiting for the other to read it; therefore they are deadlocked. In non-blocking send, a node sends a message and continues processing without verifying that the receiver node has received the message. Thus, in this section, the following are presented: 1) strategies that allow programmers to use blocking sends yet still be able exchange data in parallel on a subset of the MultiRing, and 2) a method of creating non-blocking sends by using communication assistants.

7.2.1 Blocking Send

Parallel exchange of data on a configuration means that a node sends data and receives data on the same configuration. Depending on the communication model, exchanging data in parallel may involve more than two nodes. In the pipeline communication model, a node receives data from its left neighbor and sends data to its right neighbor on a configuration. In Figure 7.2a, P_1 receives data from P_0 and sends data to P_2 on *config (8, 1)*. In the cube communication model, a node sends and receives data from the same neighbor. In Figure 7.2b, P_1 receives data from P_0 and sends data to P_0 on *config (8, 1)*. In the tree communication model, a parent node can exchange data with its left (right) child in parallel. In Figure 7.2c,P1 sends data to P_2 and receives data from P_2 on *config (8, 1)*.

To allow parallel exchange of data on *config (N, c)* using a blocking send, the following suggestions are made to ensure a sender node connects with a waiting receiver node: If either the

pipeline or cube communication model is used, the node with a one in the *(c-1)*th bit sends its data first. When a parent node wants to exchange data with a child using the tree communication model, the parent always sends its data first.

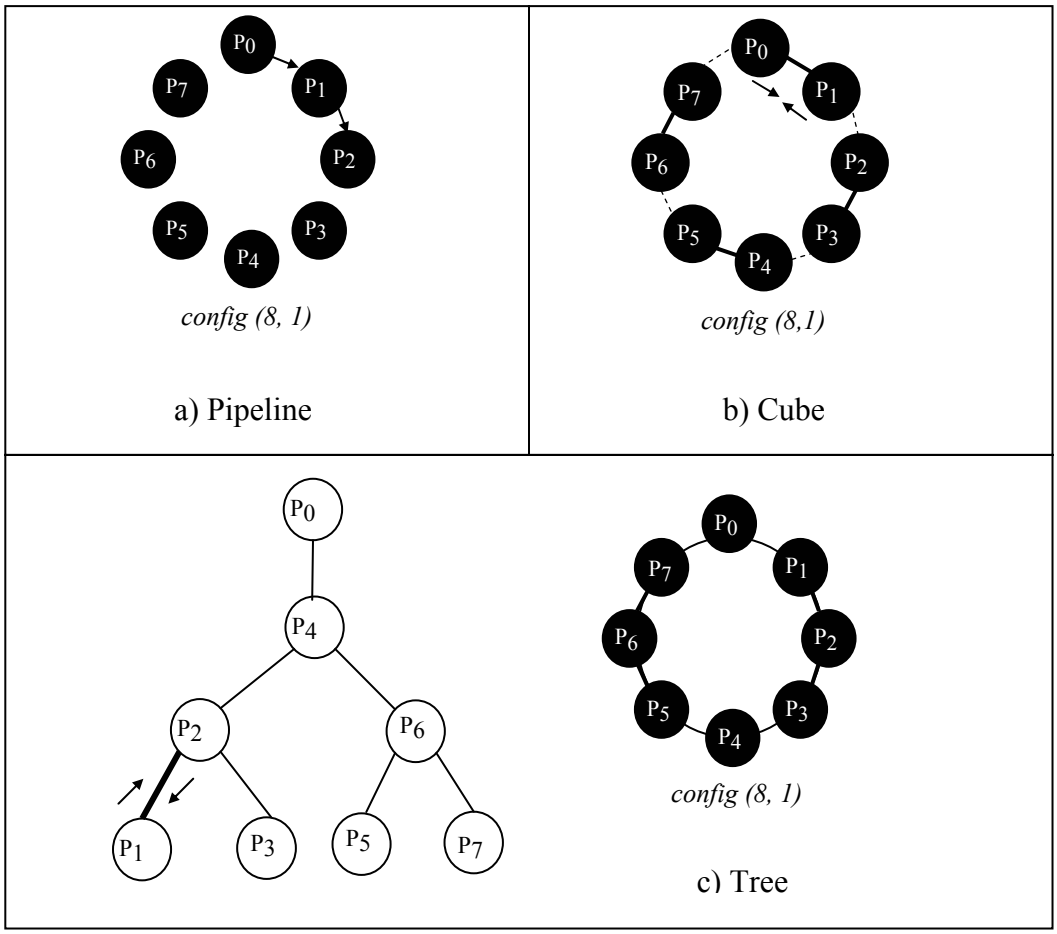


Figure 7.2: Parallel exchange of data

7.2.2 Non-Blocking Send

To implement parallel exchange of data using non-blocking send, each node has two communication assists that monitor input and output. The input assist listens to the network and

receives all messages sent to the node. The output assist is dedicated to sending messages to connected neighbors. When an application has a message to send, it writes the message to a buffer of the output assist. When an application is required to receive a message, it reads the message from a buffer of the input assist.

When the input assist receives a message from a connected neighbor it checks to see if the message has reached its destination. If it has, the message is placed in a buffer that can be accessed by the application. If the message needs to be forwarded, this message is placed in the output assist's buffer. When the message is a ring broadcast or group broadcast message, the input assist is responsible for retaining a copy of the message for the application before placing the message in the output assist's buffer.

Once the output assist notices a message in its buffer, it checks the routing table to see if the message can be sent out on the current configuration. If the message should not be sent out on the current configuration, the output assist must wait for another configuration. However, if the message can be sent out on the current configuration, the output assist is responsible for verifying that the receiver's input assist can receive the message. When a ring or broadcast message needs to be delivered, the output assist is responsible for sending multiple copies of the message at different configurations. After delivering a distribute message, the output assist is responsible for maintaining the portion of the distributed message for the application. Thus, processing a distributed message requires the output assist to write to the input assist's buffer.

Deadlock may occur when all buffers are full: the application cannot write to the full buffer in the output assist, the output assist cannot send data to any neighbor's input buffer since they are all full, and all input assist buffers are full because the application is waiting to send. In this case, after the switch has cycled repeatedly without any activity, some messages must be

dropped. Therefore, as with most parallel architectures non-blocking send may solve deadlock in most instances. However, when the system is full of messages deadlock can occur.

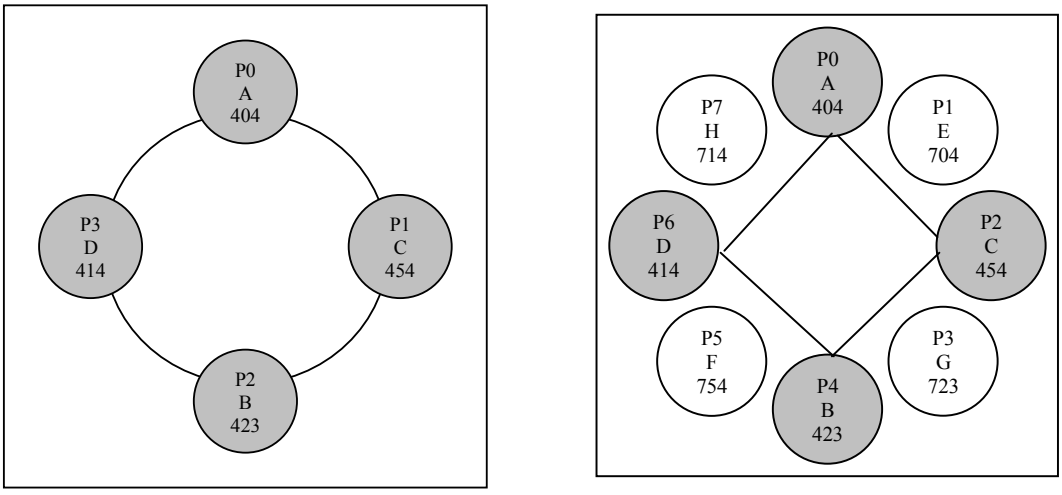
7.3 Adding nodes

Recursion is the key design element in the MultiRing switch proposed in this dissertation. Combine two rings of 2 nodes to create a ring of 4 nodes; combine rings of 4 nodes to create a ring of 8 nodes, and so forth. The new switch maintains pre-existing rings in the expanded MultiRing, thereby enabling programs that were originally implemented on a ring of N nodes to continue to use the same nodes in the new MultiRing of $2N$ nodes. This design is essential to Multiprogramming in that programs are often created for a specific group of nodes. When new nodes are added, the program still needs to run on the same network of nodes. The previous design of the switch [12] required extensive rewiring to maintain preexisting ring formations.

One added benefit of the design of the new switch is that nodes can continue to run as the switch is expanded to allow additional nodes. Before the switch is disconnected, it signals the configuration, *config (N, r+1),* which establishes N rings of 1 node where each node sends and receives messages to itself. The output assists will have to wait until the switch signals a valid configuration before continuing communication with other nodes. In essence, the programs pause communication but are still able to do local computations. After the additional switching elements have been added and the additional nodes are connected to the switch, the switch signals a new configuration and the active programs can once again communicate with other nodes.

In a network of nodes each node has a unique physical ID. This physical ID is different from the MultiRing's node ID. For example in a 4-node MultiRing machines A, B, C and D may have physical IDs of 404, 454, 423, and 414 and node IDs of P_0, P_2, P_1 and P_3 respectively.

Three main rings exist: A-B, C-D, and A-C-B-D. When additional nodes are added to the MultiRing, the existing nodes' MultiRing node ID may have to change. For example if E, F, G and H machines with physical IDs of 704, 754, 723, and 714 are added to the MultiRing, then A, B, C, and D's node IDs are modified to P_0, P_4, P_2, and P_6 respectively (Figure 7.3). Modifying the MultiRing node IDs is essential to maintaining the preexisting rings of A-B, C-D and A-C-B-D.



a) 4-node MultiRing b) 8-node MultiRing

Figure 7.3: Expanding network results in new node IDs

7.4 Costs of Multiprogramming on the MultiRing

The multiprogramming strategies described in this dissertation feature an automatic switch. In 2000, Arabnia proposed the idea of automatic reconfiguration [papers pending]; however I was heavily involved in discussions and implementations in understanding the overall impact such a novel switch would have in regards to multiprogramming on the MultiRing. The original design of the MultiRing [8, 15] featured manual requests, where a sender node sends a request to the switch for a configuration and waits until the request was granted. Manual requests

were deemed appropriate for the SPMD and SIMD algorithms running on the MultiRing. Barrier synchronization was needed to enforce lock-step communication since all communicating nodes would eventually request the same configuration. However, to implement multiprogramming with disjoint subsets of nodes, I decided the automatic switch would be best. No manual requests are made – the switch automatically changes after a set period of time and cycles through all possible configurations. However, the simplicity of the switch is not without cost.

In this multiprogramming environment, the MultiRing simulates partial configurations where independent subsets of nodes are configured differently. However in reality, the switch is globally controlled and cycles through all configurations even though no program may use them. For example, in a 16-node MultiRing, three different programs could run simultaneously: one program uses 8 nodes and requires *config (N, 2)* thru *config (N, 4)* to communicate, and the other two programs need four nodes each and only require configurations *config (N, 4)* and *config (N, 3)* to communicate. The switch for the 16-node MultiRing cycles through all configurations starting with *config (N,4)* down *to config (N, 1)* even though no program requires *config (N, 1)*.

One problem is that performance degrades for an application running on a small subset of nodes as additional nodes are added to the network. For example, if an application only needs 2 nodes, in a 4-node network it would have to wait one cycle. However with a 16-node network, this application would have to wait until 3 different configuration changes have taken place before continuing to communicate. This also applies to programs implementing a pipeline where only one configuration is needed. Future enhancements to the switch design are possible for the controller to monitor the activity and stay longer at more active configurations.

7.5 Concluding Remarks

This dissertation describes an approach to designing and developing algorithms for the MultiRing based on multiprogramming where multiple programs can execute concurrently on the same MultiRing using disjoint subsets of the MultiRing's nodes.

This chapter has shown how the switch design in Chapter 4, communication models presented in Chapter 5 and the routing operations presented in Chapter 6 work together to create a multiprogramming environment. In the next chapter, will be description of the MultiRing simulator, which was created for programmers to create and test parallel algorithms.

CHAPTER 8 : MULTIRING SIMULATOR

A programmer can create and test parallel algorithms on the MultiRing simulator that will be described next. All of the MultiRing routines a programmer needs are stored in a header file, "mr.h." Similar to what is done with PVM and MPI, the programmer must include the header file in a C program and use specific functions to initialize the MultiRing and to send and receive messages. Over 50 functions have been created including: MR_INITIALIZE (), used to establish the type of communication model, the number of nodes in the network, and the number of nodes on a ring for a problem, MR_SEND (), used to send a message from one node to another, MR_READ () used to read a message and MR_PRIGHT () and MR_PLEFT () that determine the identity of the right and left neighbors.

The MultiRing simulator is dependent on running a switch simulator and communication assistants in the background when a programmer is testing a program. The switch simulator automatically changes the configuration of the network after a specified time interval. There must be input and output communication assists for each node.

Section 1 provides descriptions of the switch simulator, communication assistants and sample applications that can run on the simulator. Section 2 contains diagrams that represent the execution flow of the simulator. In section 3, the simulator environment for multitasking windows and Unix environments is described.

8.1 Simulator description

The MultiRing simulator requires three types of processes: switch, assist, and application programs.

8.1.1 Switch

The switch is responsible for notifying all active output assists of the current configuration. In practice this can be accomplished by either message passing or shared memory. This simulator uses shared memory to broadcast the configuration of the network. Assists look at a shared buffer (actually a file named MR_CONFIG.TXT) to determine the current configuration.

The switch task repeatedly updates a configuration buffer that is read by all communication assists. It waits for a set time before changing the configuration. The configuration buffer contains two items -- the current configuration number (*1 – r*) and a status flag (*0* or *1*). The status flag is needed because even though a switch is in a configuration, there may not be time to send new data through the switch before the switch changes. The status flag is initially set to 1 at the beginning of a configuration. The switch task changes the status flag from 1 to 0 after a specified time interval to indicate that no more messages can be accepted on this configuration.

The simulator runs on a multi-tasking environment where processes run in a "time slice." Therefore, the total configuration time includes time for "all" nodes to access the configuration number and status and to send one message. On a message passing system, the time for the controller to broadcast the configuration must also be considered in determining the total configuration time.

The C program, switch.c, contains the source code for the switch task. The switch defaults to being set up for a 2-node MultiRing with a maximum configuration time of 2 seconds and an initiate time of 1 second. If MR_SWITCH.txt file is found, the default values will be overwritten by the settings in the file. However, the user is also able to pass as arguments on the

command line settings which override all other values. To start the switch for an 8-node MultiRing that has a maximum time interval of 5 seconds with 2 seconds required to send one message through the multi-stage switching network, the following command would be entered:

```
switch 8 5 2
```

8.1.2 Communication Assists

The non-blocking send operation is supported by this simulator. Input and output communication assists control the transmission messages and enable an application to send data to an output buffer and continue processing as without waiting for confirmation the data has been sent correctly.

A application communicates with its input and output assists through buffers. In this simulation, each buffer is actually a file. There are three main buffers, MR_OUT#, MR_IN# and MR_AIN#, where # is replaced by the node ID. For example P0 has buffers named MR_OUT0, MR_IN0 and MR_AIN0. When an application has a message to send, the message is written in the MR_OUT buffer. When an application is ready to read a message, it will look in the MR_IN buffer. The MR_AIN buffer is used by the input assist to filter out messages that must be forwarded to other nodes.

The output assist constantly monitors the MR_OUT buffer for new data in order to send it to one of its neighbors. In this simulation, the output assist sends a message by writing it in the MR_AIN buffer of the node it connects to on a given configuration. The input assist constantly monitors the MR_AIN buffer and when it receives a message where the destination matches its node's ID, it writes the message in its MR_IN buffer. However, if this is a message that needs to be forwarded, the input assist will write the message in its MR_OUT buffer.

Chapter 6 presented different options available for the output assist to use to decide if it should send a message. The output assist may either look up the required configuration on a routing table created during initialization or arithmetically compute the required configuration as the message arrives. In this simulation, we arithmetically compute the required configuration based on the node's ID, destination of the message and total number of nodes in the network. Once the required configuration is determined, it is compared with the current configuration in MR_CONFIG.TXT buffer. The output assist waits to send the message when the current configuration is the required one. Whether the message is written in the MR_AIN buffer of its left or right neighbor depends on the communication model.

An alternative procedure for determining if a message should be sent out on the current configuration is to first calculate the offset from the node's ID to the final destination and then apply a bit-wise mask to the offset. If the current configuration is *config(N, c)* and if there is a 1 in the binary representation of the offset in the *(c - 1)*th bit, this message can be sent out on this configuration. It is important to note that the routing options we have described are deterministic; the path a message travels is not affected by the traffic. It may be possible to alter this algorithm to include adaptive routing to avoid blocked or congested links. This could be an issue to consider for future work.

Both input and output assists are created with the same task. The C program, assist.c, contains the source code for the assist task. The assist task requires two values on the command line, the type of assist and the machine's physical ID number. If the assist is an output assist the type is 'o'; and if the assist is an input assist, the type is 'i'. Optional command line arguments include *ringN* (the number of nodes on *ringConfig* for an application) and communication model (Pipeline =0, Cube=1 and Tree =2). An assist defaults to having only 2 nodes on the main

configuration and a pipeline communication model. The assist accepts the command line arguments in the following order:

```
assist [type = i,o] [Physical ID] [Ring N] [ComModel =0,1,2]
```

To start an input assist for a node running on machine 555 for an application that requires 4 nodes and use the cube communication model for routing messages, the following command would be entered:

```
assist i 555 4  1
```

The assist looks in MR_ALLID.TXT for the complete listing of machine IDs arranged in consecutive order starting with the machine ID of P_0. For example, machine 555 is P_0 on the MultiRing if MR_ALLID.TXT contains the following machine IDs:

```
555  123  320  34  102  654  872  2029
```

8.1.3 Applications

To create a parallel application for the MultiRing Simulator, the programmer creates a C program and includes the MultiRing header file, "mr.h." All programs must include a call to MR_INTIIALIZE () and to MR_EXIT (). MR_INITIALIZE (int *argc*, char **argv* []) determines the number of nodes in the MultiRing and the MultiRing node ID, and sets it the type of communication model, and *ringN*. The arguments, *argc* and *argv* allow the user to enter the default values for node ID, *ringN* and communication model on the command line. The system defaults to have *ringN*=2 and use the pipeline communication model.

The physical machine ID of the node is stored in MR_NODEID.TXT and is read into memory if it is not passed as a value in *argv*. During MR_INTIALIZE () the file MR_ALLID.TXT is opened, the physical machine IDs of all nodes are read and the MultiRing node IDs are determined. The total number of nodes in the MultiRing is determined by the number of machine IDs in MR_ALLID.TXT. The main ring configuration is determined by the

total number of nodes in the MultiRing and *ringN*. As a final step, the head node is calculated; each ring has a head node, the node with the lowest node ID on the *ringConfig* for the problem.

Over 50 functions have been defined; however, some are private and are not available to the user. The public functions are listed below.

```
//Required Calls
void MR_INITIALIZE (int argc, char *argv[]);
void MR_EXIT();

//Functions that return status of network
int  MR_GET_COMMUNICATION_MODEL();
int  MR_GET_HEAD();
int  MR_GET_ID();
int  MR_GET_LAST_CONFIG();
int  MR_GET_RING_CONFIG();
int  MR_GET_RING_N();
int  MR_GET_TOTAL_N();
int  MR_NEXT1(int Pi, int config);
int  MR_NEXT2(int config);
int  MR_PLEFT1(int Pi, int config);
int  MR_PLEFT2(int config);
int  MR_PRIGHT1(int Pi, int config);
int  MR_PRIGHT2(int config);
void MR_SHOW_STATUS();

//Function to set communication MR_PCM, MR_CCM, MR_TCM
void MR_SET_COMMUNICATION (int com_model);

//Functions for Sending and Receiving Messages
void MR_READ(int Psource, int Pdest, int *size, int message[], int msgtype);
void MR_SEND(int Psource, int Pdest, int size, int message[], int msgtype);
void MR_SEND1(int Pdest, int size, int message[], int msgtype);
void MR_SEND_DISTRIBUTE(int size, int message[]);
void MR_SEND_GROUP(int numGroup, int size, int message[]);

//Functions for Parallel Exchange of data
void MR_PAR_EXCHANGE_WITH_CHILD(int config, int linktype,
            int ssize, int smessage[],int smsgtype,
            int *rsize,int rmessage[],int rmsgtype);
void MR_PAR_EXCHANGE_WITH_PARENT(int config, int linktype,
            int ssize, int smessage[],int smsgtype,
            int *rsize,int rmessage[],int rmsgtype);
void MR_PAR_SEND_READ1(int config, int ssize, int smessage[],int
            smsgtype, int *rsize,int rmessage[],int rmsgtype);
```

To send regular messages the programmer can define a message type that is a positive integer and use MR_SEND () or MR_SEND1 (). For sending special messages, the message type will be MR_MSG_DISTRIBUTE, MR_MSG_RING_BROADCAST or

MR_MSG_GROUP_BROADCAST. Different functions for special broadcasting were created. To send a ring broadcast message, use MR_SEND () or MR_SEND1 () and specify the message type as MR_MSG_RING_BROADCAST. Group broadcasting is application specific; the programmer must specify the number of groups. To send a group broadcast message use MR_SEND_GROUP ( ), and specify the number of groups the *ringConfig* has been divided into. Distributing a message involves sending tiles to all nodes on the *ringConfig*. To send a distribute message, use MR_SEND_DISTRIBUTE (), this function calculates the tile size automatically for even distribution of the message. After sending a distribute message, a node can simply read its input buffer MR__IN for its section of the distributed message.

Three sample programs are provided. Prog1.c demonstrates how to obtain information about the MultiRing such as node ID, number of configurations and total number of nodes on the MultiRing. Prog2.c demonstrates one node broadcasting a message and the other nodes reading it. Prog3.c prints the node ID of all nodes on the ringConfig.

```
//-----------------------------------------------
//prog1.c - access information about MultiRing
#include<stdio.h>
#include<stdlib.h>
#include "mr.h"        //required header for mr.h

int main(int argc, char *argv[])
{
      int Pi;               // processor ID
      int lastConfig;       //number of configurations of MultiRing
      int ringConfig;       //main configuration of this ring
      int ringSize;

       MR_INITIALIZE(argc, argv);

      //Use Get functions to access information about MultiRing
      Pi = MR_GET_ID();
      lastConfig = MR_GET_LAST_CONFIG();
      ringConfig = MR_GET_RING_CONFIG();   //main configuration of ring

      printf("ID: %d\n", Pi);
      printf("Number of Configurations: %d\n", lastConfig);
      printf("Ring Configuration: %d\n", ringConfig);
```

```
      //Show total number of nodes in system and head node
      printf("Number of nodes: %d\n", MR_GET_TOTAL_N());
      printf("Head node: %d\n",MR_GET_HEAD());

      MR_SHOW_STATUS();
      MR_EXIT();
      return 0;
}

//-------------------------------------------------
// prog 2.c - head node broadcasts a message to all nodes on its ring
#include<stdio.h>
#include<stdlib.h>
#include "mr.h"        //required header for mr.h

int main(int argc, char *argv[])
{     int data[] = {111,222,333}; //data contains 3 numbers
      int size = 3;   //number of data items
      int Pi, Pdest, ringSize, I, msgtype;

      MR_INITIALIZE(argc, argv);
      Pi = MR_GET_ID();

      msgtype = MR_MSG_RING_BROADCAST;
      Pdest = Pi; //Pdest is ignored when msgtype=MR_MSG_RING_BROADCAST
      if (Pi == MR_GET_HEAD())   //head node broadcasts data
      {     printf("P%d is the head node and broadcasts message\n",Pi);
            MR_SEND(Pi, Pdest, size, data, msgtype);
      }
      else  //all other nodes read data
      {     MR_READ(Pi, Pdest, &size, data, msgtype);
            printf("P%d received a message: \n",Pi);
            for (i=0; i < size; i++)
                  printf(" %d  ", data[i]);
      }

      MR_EXIT();
      return 0;
}

//-------------------------------------------------
//Prog3. c - display all id's of nodes on ringConfig
#include<stdio.h>
#include<stdlib.h>
#include "mr.h"        //required header for mr.h

int main(int argc, char *argv[])
{     int Pi, Pdest;
      int ringSize;
      int right, left;
      int ringConfig;


      MR_INITIALIZE(argc, argv);
      Pi = MR_GET_ID();
```

```
      ringConfig = MR_GET_RING_CONFIG();   //main configuration of ring
      ringSize = MR_GET_RING_N();      // number of nodes in RingConfig

      //PIPELINE COMMUNICATION MODEL
      MR_SET_COMMUNICATION (MR_PCM);  //MR_PCM, MR_CCM, MR_TCM

      //Print IDs of all nodes on ring starting from right
      printf("% d Nodes: [ P%d  ", ringSize, Pi);  //display first node
      right = MR_PRIGHT1(Pi, ringConfig);

      while (right != Pi)
      {     printf(" P%d ", right);
            right = MR_PRIGHT1(right, ringConfig);
       }
       printf(" ]\n");

      MR_EXIT();
      return 0;
}
//-----------------------------------------------
```

All of the sample programs assume the user types in the machine ID, *ringN* and communication model on the command line. Let MR_ALLID.txt contain:

```
555  123  320  34  102  654  872  202
```

For example if `prog2` is running on 4 machines using cube communication (1) and `prog3` is running on 4 machines using pipeline communication (0), the following commands would be entered.

```
prog2 555 4 1
prog2 320 4 1
prog2 102 4 1
prog2 872 4 1

prog2 123 4 0
prog2 34  4 0
prog2 654 4 0
prog2 202 4 0
```

General syntax:

```
[Program Name] [Physical ID] [Ring N] [ComModel =0, 1, 2]
```

8.2 Execution flow

In this section, execution flow of the simulator is illustrated in diagrams that include the name of private functions that the programmer will not be able to call. The purpose of these diagrams is to describe tasks that are involved when issuing a command.

Three functions a programmer can call are MR_INTIALIZE (), MR_SEND () and MR_READ (). When MR_INTIALIZE () is called, the communication model is set, a system specific initialization routine is called and ring configuration and head node are determined (Figure 8.1). The system specific initialization function is provided to allow other methods of initiating nodes to be created for future work.

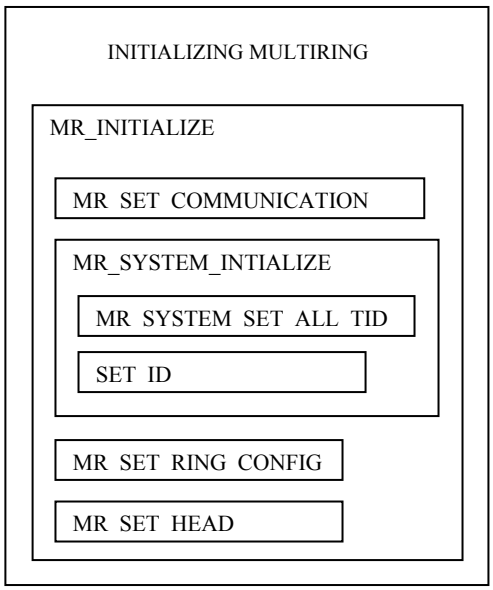


Figure 8.1: MR_INITIALIZE

When MR_SEND () is called, the message is packed before it is sent to the MR_OUT buffer in the assist (Figure 8.2). Currently only arrays of integers are accepted as messages. Packing a message puts the size in the message at element 0.

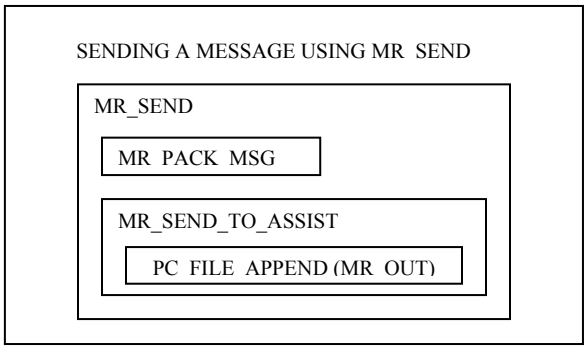


Figure 8.2: MR_SEND

A call to MR_READ() initiates a call to a system specific read that reads the input buffer (Figure 8.3). In our simulation, the file MR_IN is read, and the message returned is sent to MR_UN_PACK to extract the data from the message.

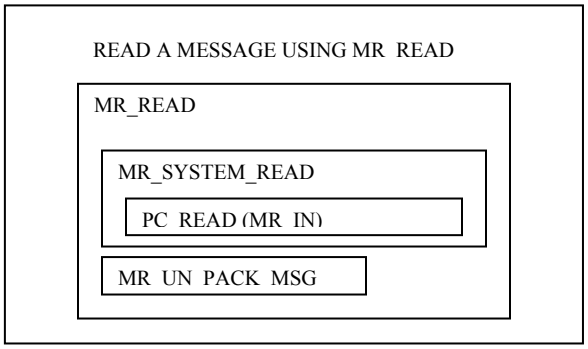


Figure 8.3: MR_READ

The communication assist calls ASSIST_INITIALIZE () to set the communication model, determine the total number of nodes, set the node ID, ring Configuration and open a buffer for reading (Figure 8.4). For an input assist, MR_AIN# is opened for reading; however, for an output assist, MR_OUT# is opened, where # is replaced by the node ID.

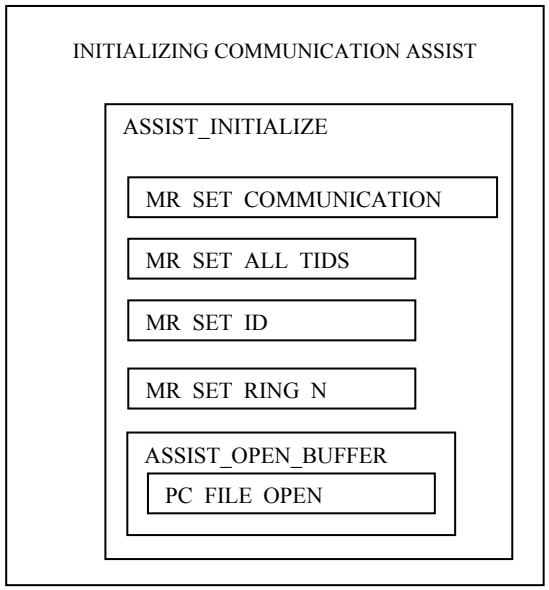


Figure 8.4: ASSIST_INITIALIZE

The assist process loops continuously and calls ASSIST_READ () followed by a call to ASSIST_SEND (). Inside each function there are different options based on the type of assist and the type of message. For example, if an output assist processes a regular message (Figure 8.5), in ASSIST_SEND, it loops up the required configuration and checks the status of the current configuration before sending the message to MR_AIN of the connected neighbor. However, if the output assist receives a special message, the assist may have to call a function dedicated to sending copies of the message out on more than one configuration depending on the configuration model used. For example in Figure 8.6, a broadcast message may go either to ASSIST_INITIATE_BROADCAST () or ASSIST_FORWARD_BROADCAST ().

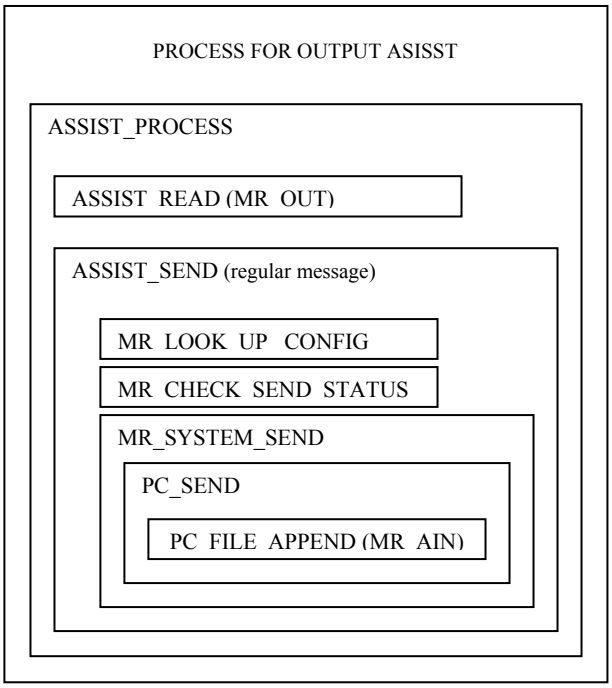


Figure 8.5: Output assist sends a regular message

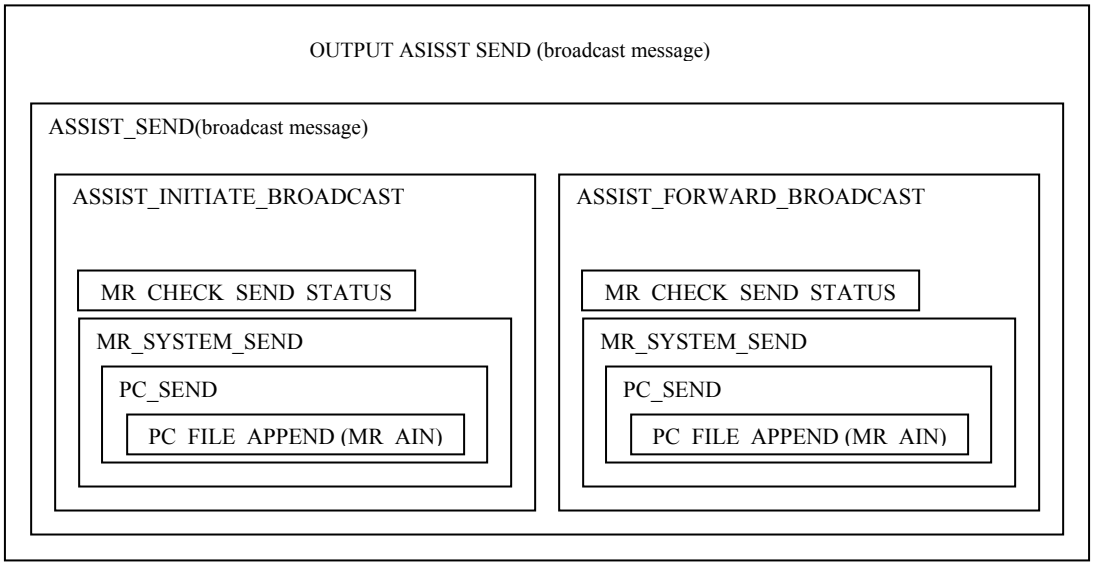


Figure 8.6: Output assist sends a broadcast message

An input assist's main process is to read a message from MR_AIN and write it to MR_IN if the message has reached its destination or to MR_OUT if the message needs to be forwarded

(Figure 8.7). Special messages, like broadcasting messages, may require writing to both MR_IN and MR_OUT (Figure 8.8).

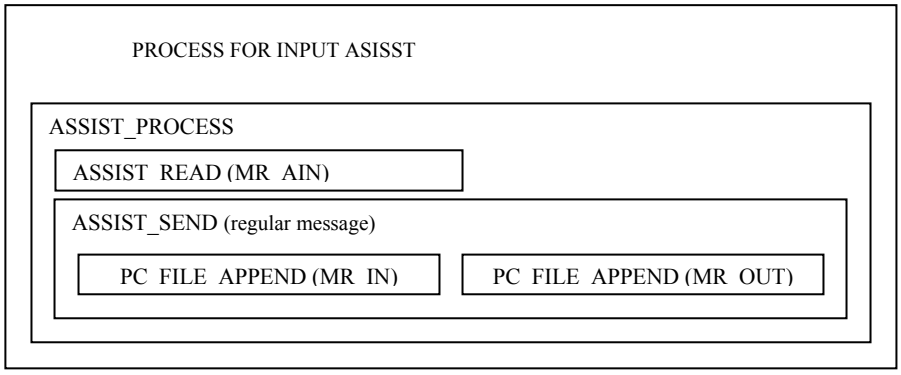


Figure 8.7: Input assist receives a regular message

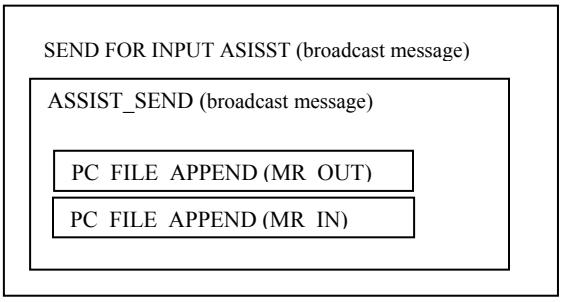


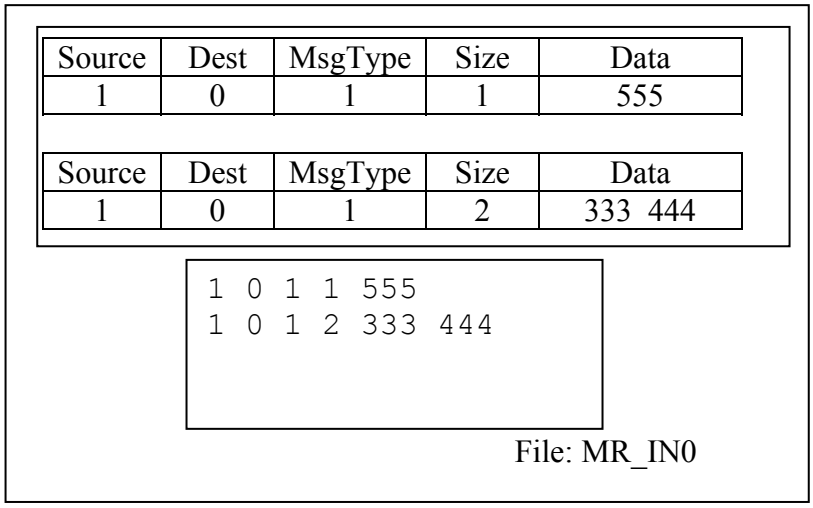
Figure 8.8: Input assist reads a broadcast message

8.3 Simulator Environment

The simulator has been designed to send messages that are arrays of integers. Sending one integer requires creating an array and storing the integer in element 0.

```
int data[] = {555}; // store message 555
```

The header for the message includes source ID, destination ID, message type and data size. In Figure 8.9, two messages are stored in the input buffer of P0. One message sent from P1 contains one integer, 555. The other message set from P1 contains two integers 333 and 444.



Source	Dest	MsgType	Size	Data
1	0	1	1	555

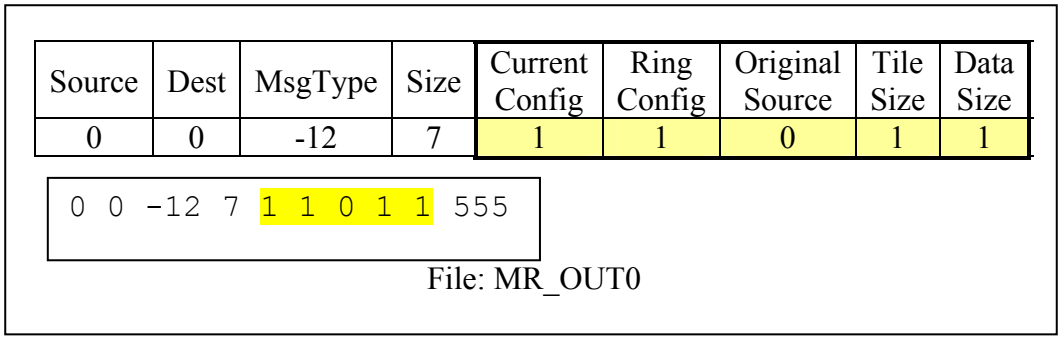
Source	Dest	MsgType	Size	Data
1	0	1	2	333 444

```
1 0 1 1 555
1 0 1 2 333 444
```

File: MR_IN0

Figure 8.9: Regular Messages

A special header is added for ring broadcast, group broadcast and distributed messages. The current configuration, ring configuration, original source and tile size are needed to help route the message. In Figure 8.10, a ring broadcast message of 555 is placed in P0's MR_OUT buffer. (A ring broadcast message type is defined as -12.)



Source	Dest	MsgType	Size	Current Config	Ring Config	Original Source	Tile Size	Data Size
0	0	-12	7	1	1	0	1	1

```
0 0 -12 7 1 1 0 1 1 555
```

File: MR_OUT0

Figure 8.10: Broadcast Message

There are many files that are needed by the simulator. Table 8.1 contains a description of the files.

Table 8.1 Simulator Files

FILE TYPE	FILENAME	FILE DESCRIPTION
BUFFERS	MR_OUT#	Output Buffer
	MR_IN#	Input Buffer
	MR_AIN#	Input Assist's Internal Buffer
ENVIRONMENT FILES	MR_NODEID.TXT	Physical Machine ID [555]
	MR_CONFIG.TXT	Current Configuration of MultiRing [1 0]
	MR_ALLID.TXT	All machine IDs in MultiRing arranged in order starting with P0
	MR_LAYOUT.TXT	ALL machine IDs in MultiRing arranged in order of connections to switch.
	layout.c	Converts Layout.TXT into MR_ALLID.TXT
HEADER FILES	mr.h	Contains MultiRing Functions
	mrFile.h	List of Environment Files
	gcwtime.h	Timing functions for switch
	PCassist.h	Assist Helper functions
	PCbuffer.h	All file IO routines to access buffers
UNIX UTILITY FILES	killassit.c	Kill background tasks
	doubleit.c	Used when initializing assists
	runtest.sh	Shell program that starts assists, and test program
	stoptest.sh	Shell program that stops assists, switch and test programs
	results.sh	Shell program that displays data stored in all buffers
ASISST	assist.c	Assist program
SWITCH SIMULATOR	switch.c	Switch simulator

8.4 Concluding remarks:

The MultiRing simulator described above was created to test the routing and communication strategies. This simulator is useful in that other programmers can use it to create and test parallel algorithms.

Three different parallel sorting algorithms are presented in the next chapter that can run on the MultiRing simulator. These sorting algorithms will incorporate ring broadcasting, group broadcasting and distributing.

CHAPTER 9 : SORTING ON THE MULTIRING

Sorting, commonly used in a variety of computing applications, consists of ordering keys each of which is associated with an item. Examples of sorting include alphabetizing a database of consumer information by customer name, preparing a list of inventory at a warehouse by product ID, and listing checks on a bank statement by either check number or date check was cashed. Sorting can be time consuming if there are a lot of keys to organize. Instead of a using a single node to sort a large list of keys, one can use a collection of nodes that work together in parallel to sort the keys where each node sorts a subset of the total number of keys.

Quicksort is just one out of a variety of serial sorting algorithms. In quicksort, a median key is selected and is used to separate the keys into two groups: one group contains keys that are less than or equal to the median key and another group contains keys greater than the median key. Quicksort is applied recursively to each group until each group contains at most one item.

In this chapter, three parallel sorting algorithms are presented, bitonic sorting, MultiQuicksort and bin-collecting. All algorithms presented begin at the collecting phase – at this phase, all keys have been evenly distributed among the nodes and sorted locally using quicksort. If there are M keys and N nodes, each node has M/N keys already in sorted order. This chapter focuses on how the keys are collected at different stages until all keys are in "sorted order." The term "sorted order" for bitonic sort has a different meaning than it has for MultiQuicksort and bin-collecting. With bitonic sorting, a set of keys is considered sorted if all M keys are in sorted order on each node. However, with MultiQuicksort and bin-collecting, a set of keys is considered sorted if the keys within each node are sorted and the largest key in node P_i is less than or equal to the smallest key in node P_{i+1} for $0 \leq i < N-1$.

Bitonic, MultiQuicksort and bin-collecting are based on algorithms developed for the hypercube; therefore the cube communication model (Figure 6.3) is used for routing and broadcasting messages. These algorithms illustrate how ring broadcasting, group broadcasting and distributing (Chapter 6) can be implemented in a program.

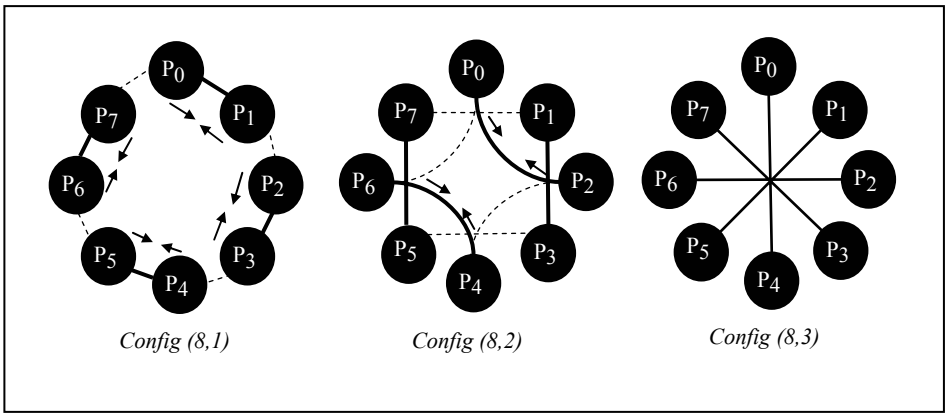


Figure 9.1: Cube communication on a MultiRing

9.1 Bitonic Sorting

The collecting phase of bitonic sorting, [55, 64] involves repeatedly merging sorted lists of keys into larger and larger lists preserving the sorted order. In the end, each node contains all of the *M* keys in sorted order. If *L* nodes are used on an *N*-node MultiRing to collect all *M* keys, bitonic sorting requires configurations of the network starting with *config (N, r)* and ending with *config(N, ringConfig)* where, $N=2^r$, $L=2^x$, $r >= x$ and $ringConfig = r - x + 1$. With the cube communication model, nodes are grouped in pairs for exchanging lists, so on each configuration, a node, P_i sends a copy of its sorted list to its neighbor, P_{next}, and receives a copy of P_{next}'s sorted list. After P_{next}'s list is received, both lists are merged into one complete list. The network is reconfigured and each node shares its new list with its neighbor on the new configuration. The

lists double in size during each configuration of the network until in the last configuration, P_i sends and receives a list with *M*/2 elements. As a result, merging takes longer to perform in successive configurations.

```
// BITONIC_SORT() – COLLECTING PHASE
//Reconfigure the network until all lists have been merged
for (config = r; config >=ringConfig; config--)
{  Pnext = nextNode(config) ;    //Find neighbor using cube communication model
   //Read and write on the same configuration
   PAR
      Read(Pnext, Pi, recvList);  //Read neighbor's list
      Send(Pi, Pnext, List);      //Send Pi's list to neighbor
  //Merge recvList into List—List is updated to include keys in recvList
   Merge(List, recvList);
}
```

Figure 9.2- Figure 9.4 illustrate the steps needed for a bitonic sort of 32 keys on an 8-node MultiRing when all 8 nodes are used for sorting. The keys are dispersed evenly among the nodes each of which originally contains a sorted list of four keys. In the example, P_0 contains a list with 4, 7, 8 and 11; P_1 contains a list with 3, 10, 21 and 31; P_2 contains a list with 1, 15, 16 and 18; P_3 contains a list with 12, 22, 25 and 28, P_4 contains a list with 6, 17, 23 and 27; P_5 contains a list with 2, 5, 13 and 19; P_6 contains a list with 0, 9, 14 and 20, and P_7 contains a list with 24, 26, 29 and 30.

When 8 nodes are used, three configurations are needed to complete the collecting phase of the bitonic sort. In step 1, the network is set to *config(8,3)* and lists are exchanged and merged

with the neighbor's list (Figure 9.2). In step 2, the network is arranged in *config(8,2)* and lists are exchanged and merged (Figure 9.3). Finally in step 3, the network is arranged in *config(8,1)* and lists are exchanged and merged with the neighbor's list (Figure 9.4). After bitonic sorting, all nodes contain all 32 keys in sorted order.

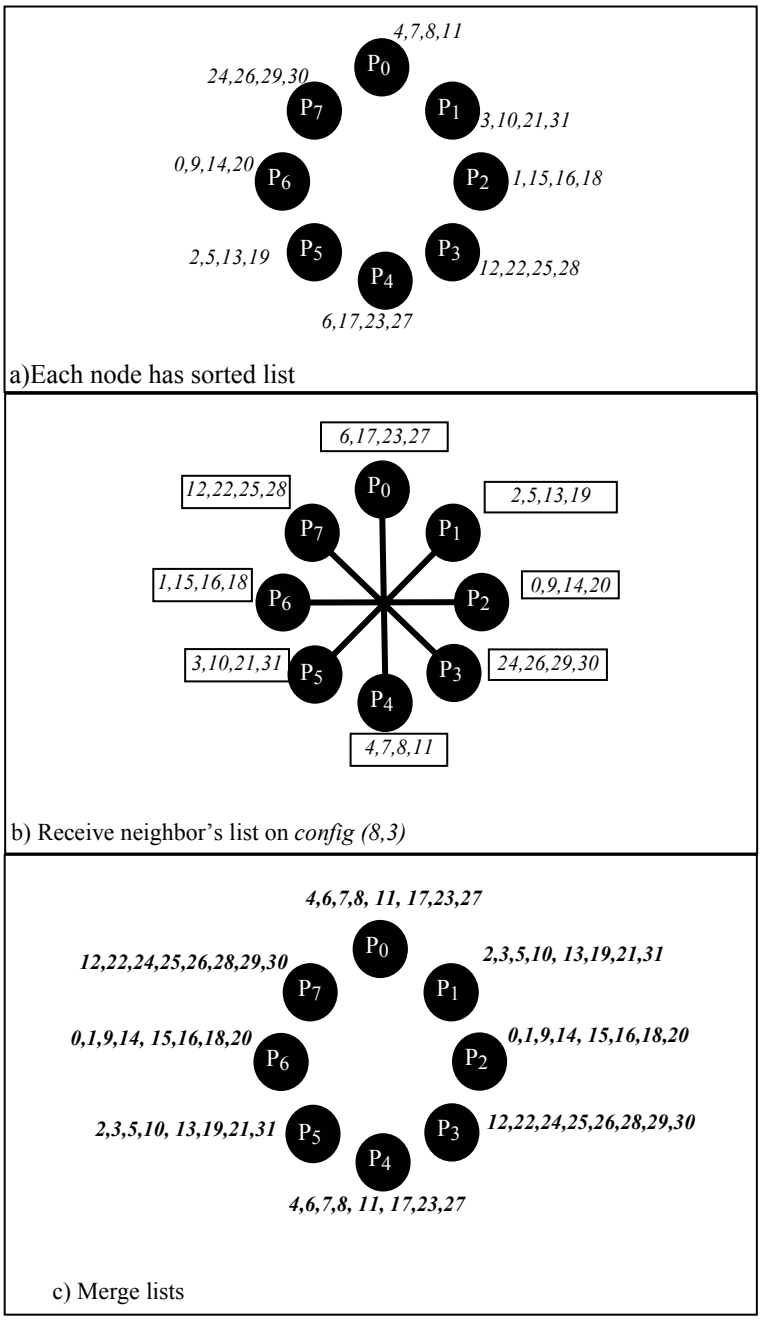


Figure 9.2: Bitonic Sort [Step 1 of 3]: Merging list with neighbor's list on *config(8,3)*.

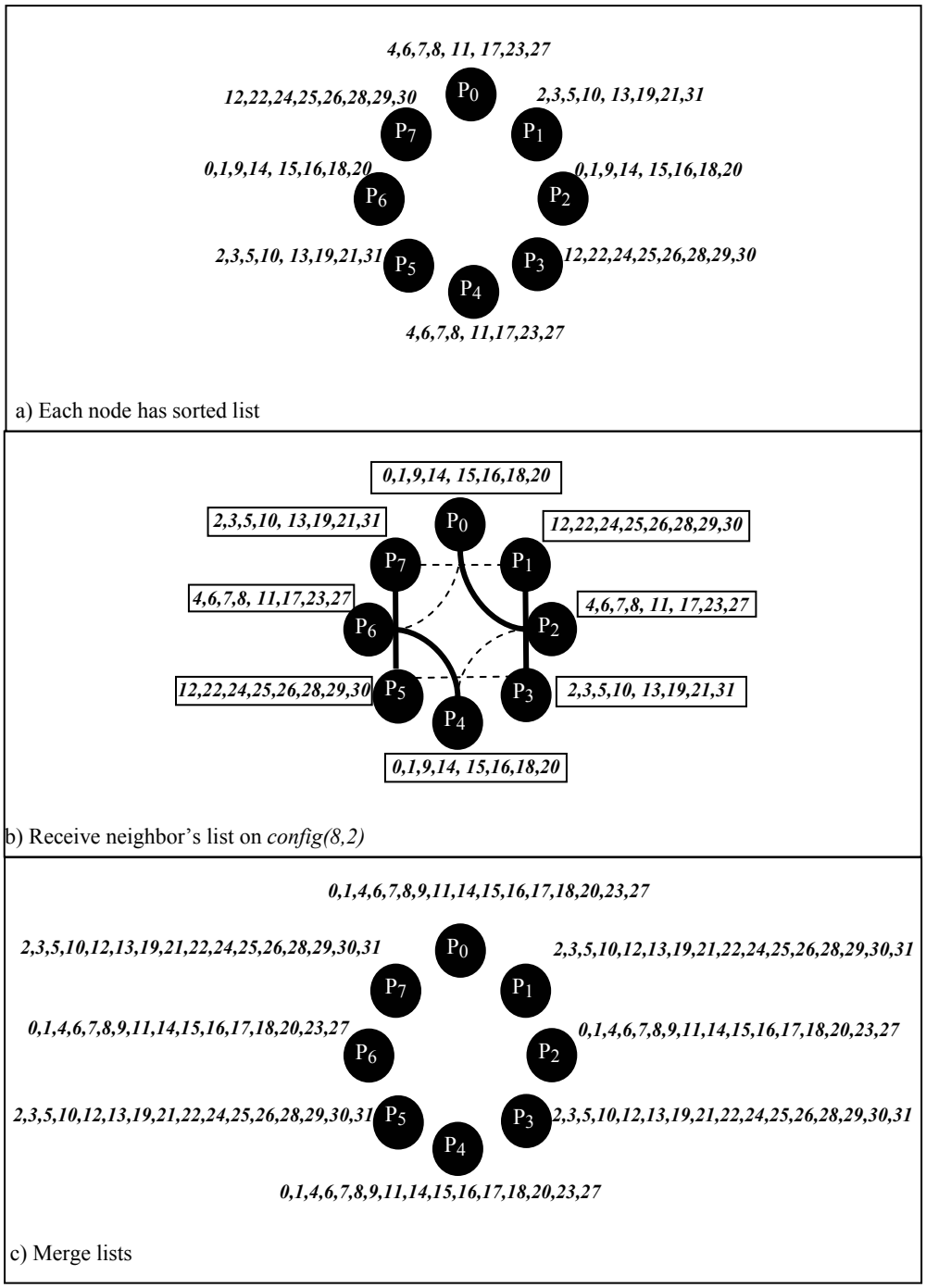


Figure 9.3: Bitonic Sort [Step 2 of 3]: Merging list with neighbor's list on *config(8,2)*.

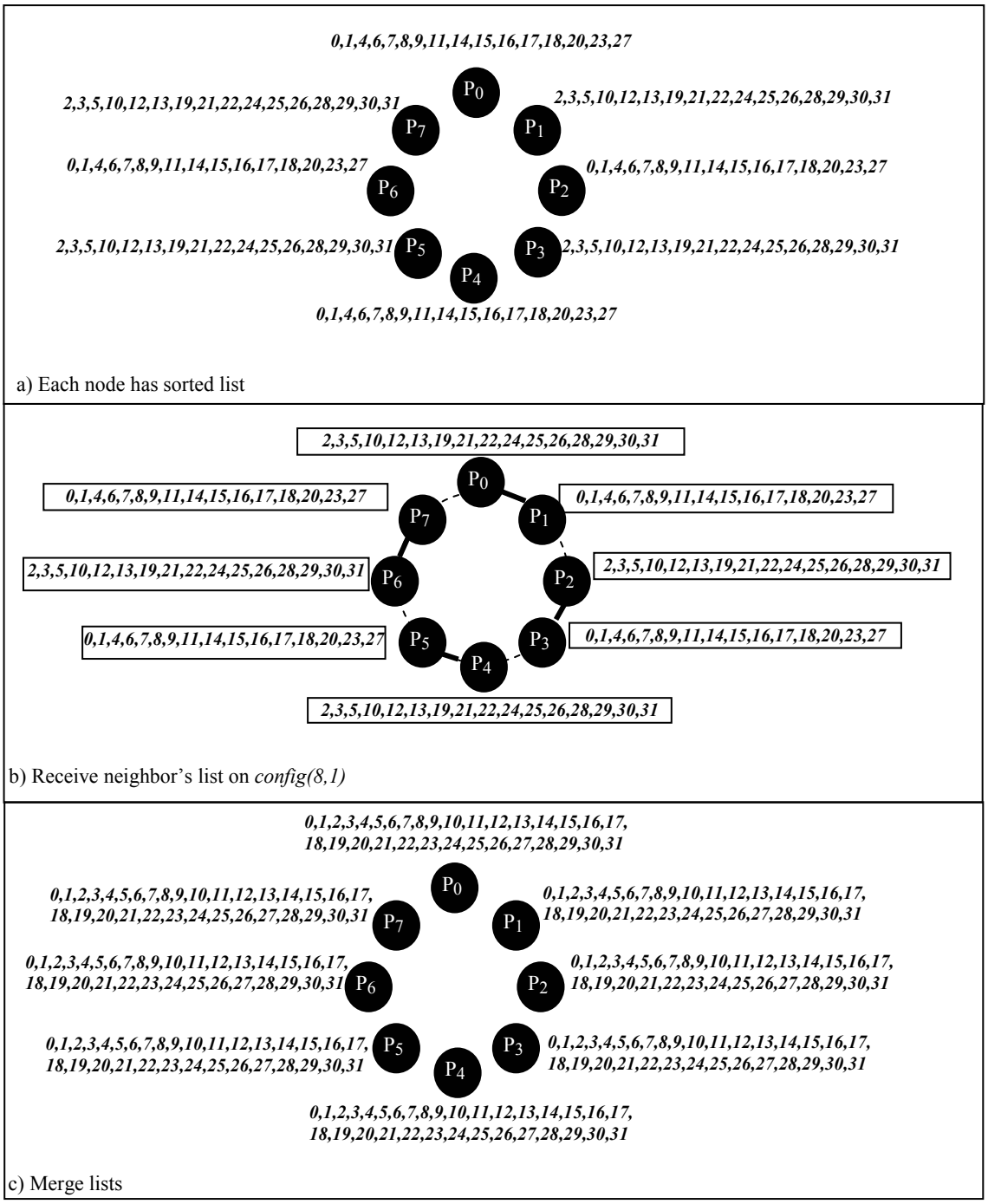


Figure 9.4: Bitonic Sort [Step 3 of 3]: Merging list with neighbor's list on *config(8,1)*.

9.2 MultiQuicksort

The MultiQuicksort is based on the *HyperQuicksort* described by Wagar [64] for use on the hypercube. This parallel algorithm mimics the behavior of the quicksort by having a median

key, *K*, to divide the keys into two lists: those whose keys are greater than *K* and those whose aren't.

In addition to exchanging lists with neighbors, the MultiQuicksort algorithm incorporates group broadcasting, a routing operation that was described in Chapter 5. As with bitonic sorting, if *L* nodes are used on an *N*-node MultiRing, MultiQuicksorting, in order to exchange lists among neighbors, requires exchange configurations starting with *config (N, r)* and ending with *config(N, ringConfig)* where, $N=2^r$, $L=2^x$, $r >= x$ and $ringConfig = r - x + 1$. However, additional reconfigurations are needed to broadcast a median key to all nodes in a group for all exchange configurations.

The algorithm for MultiQuicksort is as follows:

1. Form one group of *L* nodes; *numGroups* is set to 1. If $L = N$, the nodes will be numbered from P_0 to P_{N-1} on an *N*-node MultiRing. However, if $L < N$, the group of *L* nodes is formed with only nodes on the ring on *config (N, ringConfig)*.
2. The first configuration of the network for exchanging lists will be *config(N, r);* therefore initialize *exchangeConfig* to *r*.
3. Repeat until there are *L* groups (i.e. Loop while *numGroups* <= *L*/2)
   a) Let the node with the lowest node ID number within each group find its median key, *K*, and broadcast this key to all nodes in its group.
   b) Each node separates its keys into two lists according to the median key. The *lower list* contains all keys less than or equal to *K* and the *upper list* contains all keys greater than *K*.

c) Split each group of nodes arranged in numerical order into two sections, lower section and upper section. The first half of the nodes belong in the lower section, the remaining nodes belong in the upper section.

d) Arrange the network into *config(N, exchangeConfig)*. Using the cube communication model, nodes communicate in pairs; on *config(N, exchangeConfig),* P_i exchanges data with P_j where $i < j$ and P_i is a member of the lower section and the P_j is a member of the upper section. The node with the smaller ID, P_i, sends its *upper list* to the node with the larger ID, P_j, and P_j sends its *lower list* to P_i.

e) Each node merges the new list it receives with the appropriate section and deletes the section it sent from its local list.

f) Update *exchangeConfig* to be ready for the next configuration for exchanging lists:

```
exchangeConfig--
```

g) Let the lower and upper sections of each group form two distinct groups. The number of groups doubles:

```
numGroups = numGroups * 2.
```

Only the node with the lowest ID in the group P_{low} determines the key for splitting the list used by all nodes in the group. Node P_i uses the current number of groups the network is arranged in and its node ID to determine P_{low} for its group:

```
NODEID  GetLowNode (NODEID Pi, int numGroup)
{  NODEID Plow;
   //Find Plow on ring
    int groupSize = L/numGroups;   // L= number of nodes on ringConfig
    Plow = ((int)Pi/ (int)groupSize) * groupSize;
```

```
    Return Plow;
}
```

P_{low} determines *K*, the median key -- the key located in the middle of the list: (*K = List[ListSize/2])*. P_{low} then broadcasts *K* to the rest of the nodes in the group using group broadcasting. All nodes separate their keys into two lists using *K*. Since the keys are already sorted, a binary search can be used to find the key to divide the lists.

```
//MULTIQUICKSORT() – COLLECTING PHASE
exchangeConfig = r;
for (numGroups =1; numGroups >= N/2;  numGroups=numGroups*2)
{     //Find Plow on ring
     Plow = GetLowNode(N,numGroup);
      //Plow broadcasts median key
     if (Plow == Pi)
      {     //Get Median Key
            K = list[numList/2];
            //Broadcast Key to others in group
            GROUP_BROADCAST(Pi, exchangeConfig, ringConfig, K);
      }
      //All other nodes read the key
      else
            READ(Plow, Pi, 1, K);
      //Divide list into 2 groups based on key K
      Divide(list, upperList, lowerList, K);
      //Find ID of neighbor on ring
      Pnext = NextNode(exchangeConfig);
```

```
      //Send/Read partial List to/from Pnext & Merge
      if (Pi > Pnext)   //Pi holds the larger numbers
      {  PAR
            SEND(Pi, Pnext,  lowerList);
            READ(Pnext, Pi,  recvList); //receives Pnext's upperlist
         Merge(list, upperList, recvList);
      }
      else  //Pi holds the smaller numbers
      {  PAR
            SEND(Pi, Pnext,  upperList);
            READ(Pnext, Pi,  recvList); //receives Pnext's lowerlist
         Merge(list, lowerList, recvList);
      }
      exchangeConfig --; //update exchangeConfig
  }//end for
}
```

To group broadcast, P_{low} can either send individual messages to each node in its group or initiate a group broadcast message.

The algorithm for group broadcasting sending individual messages is given below:

```
//Plow Broadcasts the median Key
If (Pi == Plow)
{     Pnext = Plow;
      //GROUP_BROADCAST --sending individual messages to each node in its group
      For (count =1; count < groupSize; count++)
      {     Pnext = NextNode (Pnext, exchangeConfig)
```

```
            Send (Plow, Pnext,  K);
      }
}
Else //All other keys reads one message
      Read (Plow, Pi, 1, K)
```

If P_{low} initiates a group broadcast using a special GROUP_BROADCAST flag, the Read operation involves forwarding the special message to other nodes:

```
If (Pi == Plow) //Plow sends the key once on each required configuration
{
      //GROUP_BROADCAST – initiating a group broadcast
      For (config = exchangeConfig; config >ringConfig; config++)
       {
            Pnext = NextNode(Pi, config);
            Send (Plow, Pnext,GROUP_BROADCAST, K);
      }
}
Else
{     //READ involves forwarding the group message
      Read(Plow, GROUP_BROADCAST,  K); // K arrives on groupConfig
      For ( config= groupconfig; config > ringConfig; config++)
      {
            Pnext = NextNode(Pi, config);
            Send (Plow, Pnext,GROUP_BROADCAST, K);
      }
}
```

Figure 9.5- Figure 9.7 illustrate the steps of the collecting phase of a MultiQuicksort using 8 nodes to sort 32 keys on an 8 node MultiRing. Each node originally contains four keys in sorted order -- the same initial state used for the bitonic sort example. In Figure 9.5 all 8 nodes are organized in a single logical group. P_0, P_{low} for the group, broadcasts its median key, *7*. All other nodes divide their list into lower (shaded keys) and upper lists based on *7*, exchange partial lists on *config(8,3)* and merge received lists. In Figure 9.6, the nodes are divided into two groups: P_0-P_3 are in one group and P_4 –P_7 are in another group. P_0 is the low of its group and P_4 is the low of its group. Each P_{low} broadcasts its median key to its group, and lists are divided and exchanged on *config(8,2*). In Figure 9.7, the nodes are divided into four groups. P_0, P_2, P_4 and P_6, P_{low}s of their group, broadcast their median key. The lists are divided, exchanged on *config (8,1)* and merged.

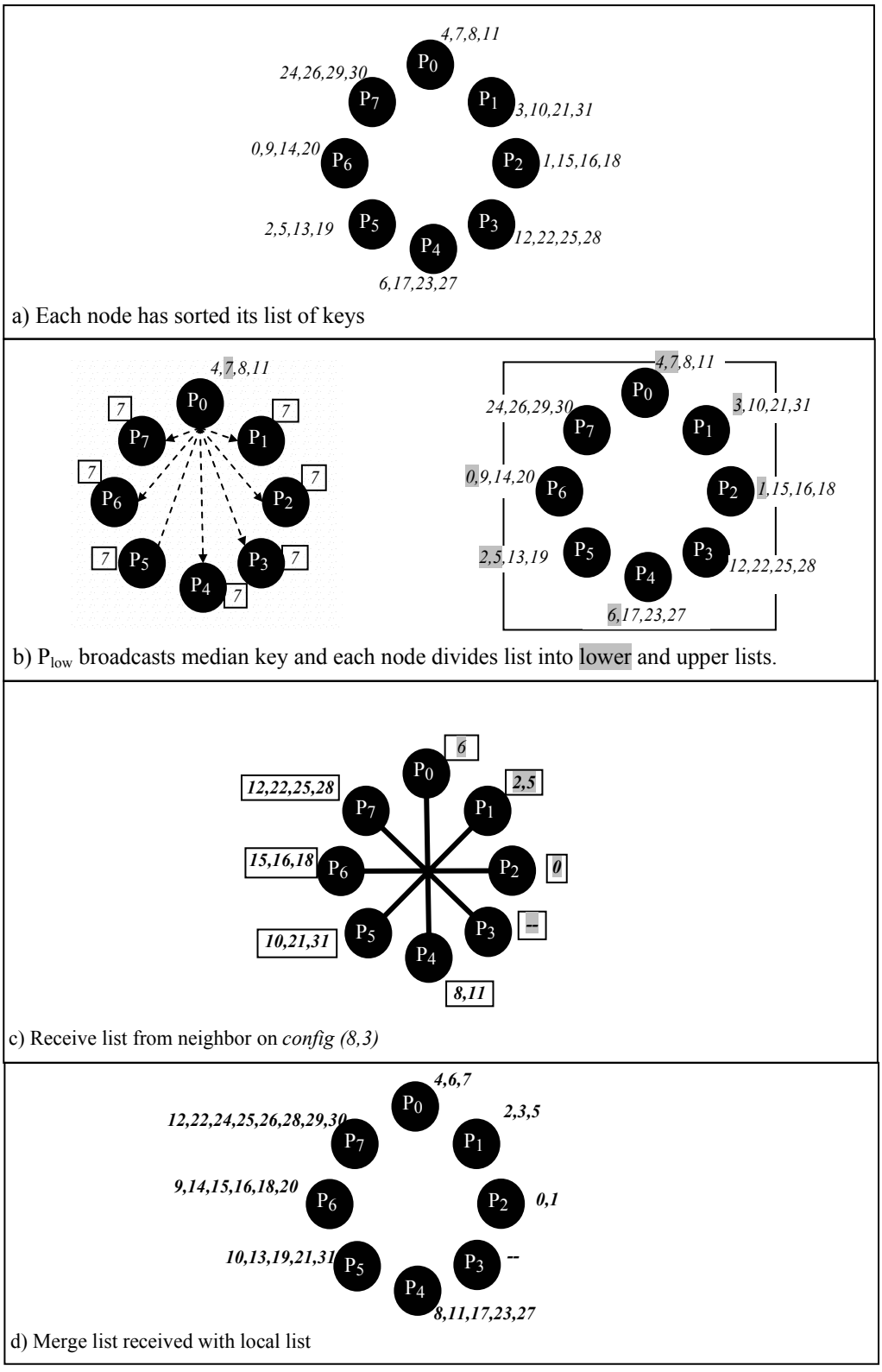


Figure 9.5: MultiQuicksort [Step 1 of 3]: One group of 8 nodes

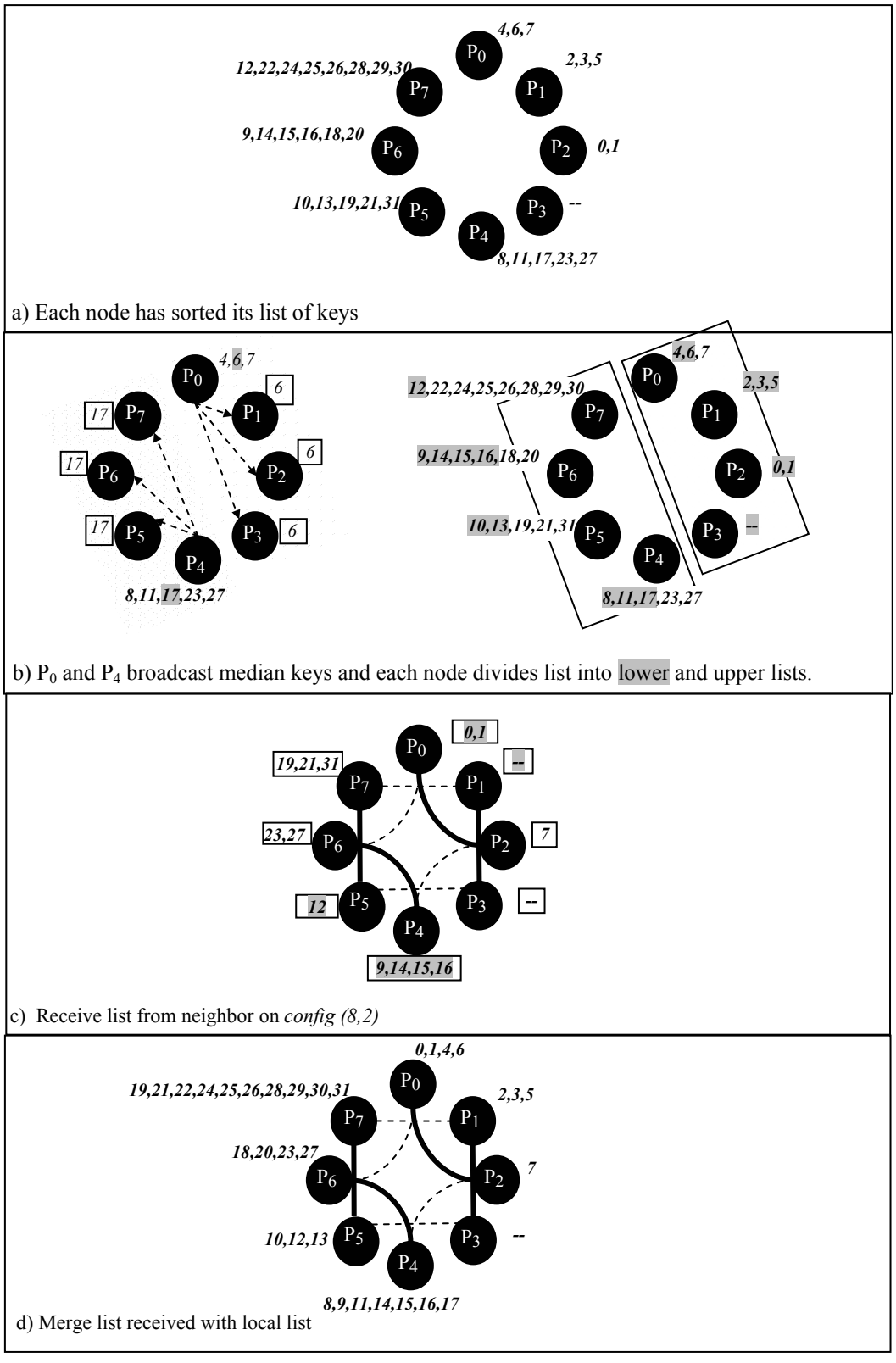


Figure 9.6: MultiQuicksort [Step 2 of 3]: Two groups of 4 nodes

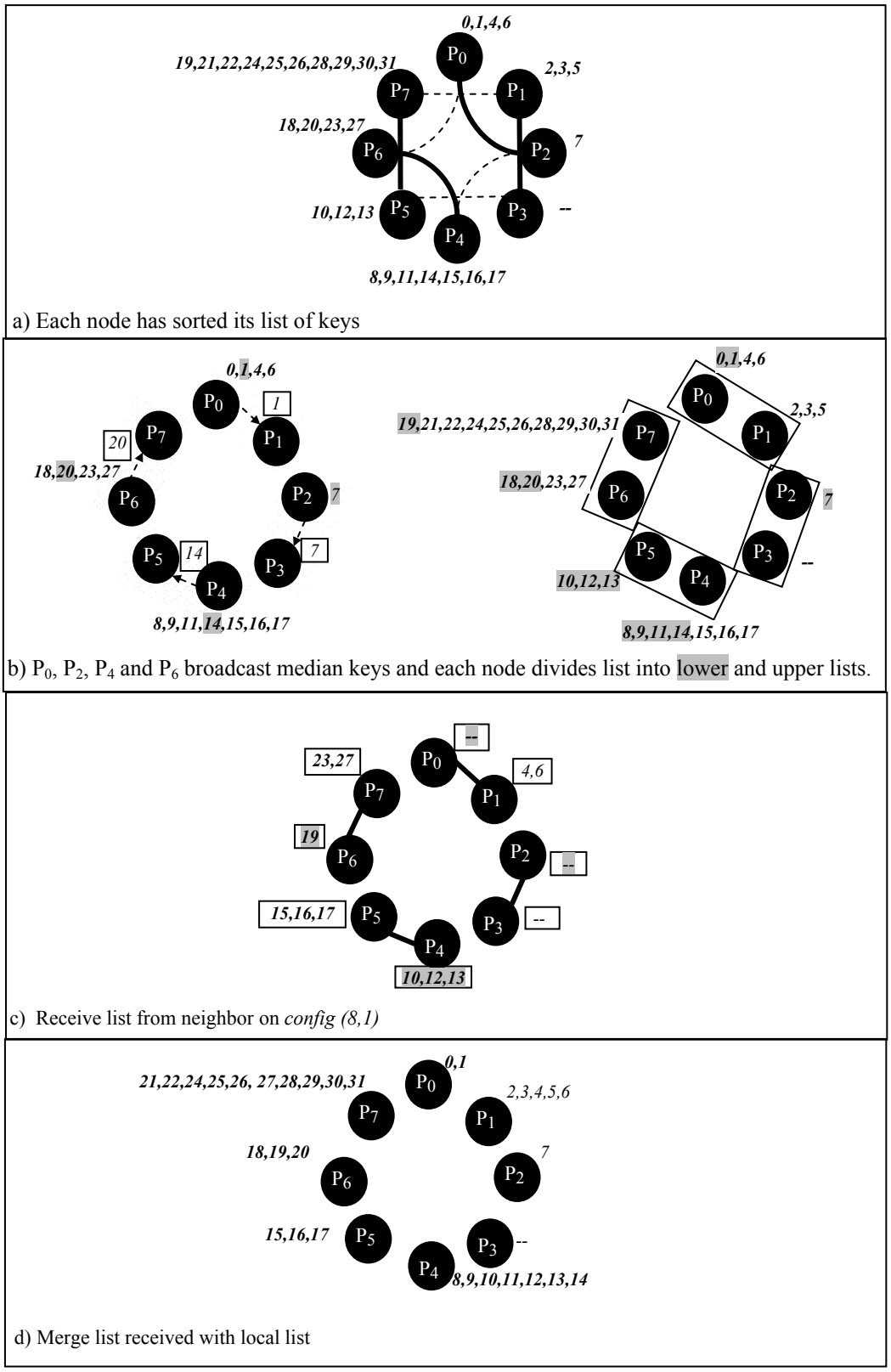


Figure 9.7: MultiQuicksort [Step 3 of 3]: Four groups of 2 nodes

After the final stage, all the keys that a node contains are in sorted order. With MultiQuicksort, some nodes may not contain any keys – in this example P_3 doesn't contain any keys in the end. However, the keys are considered in sorted order since the largest key in P_i is smaller that the smallest key in P_j for all $i < j$.

9.3 Bin-Collecting

Bin-collecting combines the advantages of bitonic sort (minimal number of required communication between nodes) and MultiQuicksort (small average length size of each message). Bin-collecting is similar to Seidel and George's [55] parallel *binsorting 2* algorithm for the hypercube. This parallel algorithm is based on separating *M* keys into *N* bins and requiring each node to collect the keys in only one bin. In general, node P_i is responsible for collecting the keys in bin *i*.

A splitting key, `binkey[i]` is a term used to describe the maximum key for bin *i*. All keys less than or equal to `binkey[i]` belong in bin *i*, and *N-1* splitting keys are needed to divide the original list in to *N* bins. The splitting keys are selected from a random sample of the *M* keys. Each node P_i separates its original list into *N* bins using the splitting keys and merges all *i*-bins from all other nodes in the network, maintaining the sorted order.

In bin-collecting, a node has keys in *N-1* different bins to send to the other nodes. A complete exchange of messages occurs when a node P_i has a distinct message $M_{i,j}$ for every node P_j in the network, and node P_i sends $M_{i,j}$ to each node P_j and receives $M_{j,i}$ from each node P_j. *N*-1 different messages can be sent where each message contains the contents of one bin; however bin-collecting sends composite messages (described by Seidel and George) which contain a collection of bins.

The goal of the bin-collecting algorithm is to have minimal communication between nodes and small average size messages. Using bins to organize keys help reduce the size of messages. Each node divides its original list of keys into different bins using splitting keys. If the distribution of keys is known, it is easy to designate the splitting keys for each bin. For example, in sorting numbers from 1 to 40 when each number is listed only once, three splitting keys,10, 20 and 30, are used if 4 nodes are used for sorting. This ensures that the bins contains the same number of keys; in this example, each bin contains 10 keys. However, if the distribution of keys is unknown, the splitting keys are generated from a sample of keys collected from each node.

P_0 collects the splitting keys and uses ring broadcasting to broadcast the splitting keys for the different bins to all other nodes. Each node separates its list into bins using the splitting keys. Node P_i collects *i*-bins from all other nodes. As it receives the bins, it merges the keys maintaining the sorted order of the keys. Node P_i sends all *j*-bins to P_j for all nodes where $j \neq i$. When P_i sends a bin, it deletes this bin from its collection.

As with the other sorting methods described in this chapter, if *L* nodes are used on an *N*-node MultiRing, MultiQuicksort requires configurations starting with *config (N, r)* and ending with *config(N, ringConfig)* where, $N=2^r$, $L=2^x$, $r >= x$ and $ringConfig = r - x + 1$ for exchanging lists with neighbor

The algorithm for bin-collecting is as follows:

1. To produce a sample of keys, each node selects at least one key from its lists of sorted keys. In our example the median key is selected, but it is also possible to select minimum and maximum keys or send *s* randomly selected keys.

2. P_0 collects the sample keys from each node creating a sample list of *sL* keys. This list is sorted and *L-1* splitting keys are chosen for the *L* bins. P_0 broadcasts the splitting keys to all nodes on *ringConfig* using ring broadcasting.
3. The list is separated into *L* bins using the splitting keys.
4. Node P_i collects all *i*-bins from all other nodes in the network:

For each *exchangeConfig*, starting with *config(N, r)* down to *config(N, ringConfig)*

a) Each node separates its collection of bins into two lists. The *lower list* contains the lower half of the collection and the *upper list* contains the upper half of the collection.

b) The network is arranged into *config(N, exchangeConfig)*. Using the cube communication model, nodes communicate in pairs; on *config(N, exchangeConfig),* P_i exchanges bins with P_j where $i < j$. The node with the smaller ID, P_i, sends its *upper list* to the node with the larger ID, P_j, and P_j sends its *lower list* to P_i.

c) Each node merges the new bins it receives with the appropriate section and deletes the bins it sent from its local collection.

d) *exchangeConfig* is updated to be ready for the next configuration for exchanging lists:

```
exchangeConfig--
```

e) The number of bins in the local collection is reduced by half:

```
numBins = numBins/2
```

```
//BIN-SORTING – COLLECTING PHASE
```

//Each node chooses a sample key – the median key

```
K = list[numList/2];
```

//P0 broadcasts receives all sample keys

```
if (P0 == Pi)
{     binKeys[0] = K; //store my sample key
      //Read N-1 keys from all other nodes
      for(i=1; i < N; i++)
          Read(binKeys[i]);
      //Sort Sample
      QuickSort(binKeys);
      //Broadcast Key to others in group
      groupConfig = r;
      GROUP_BROADCAST(Pi, groupConfig, ringConfig, binKeys);
}
//All other nodes send single sample key and receive all sorted samples
else
{     Send(Pi, P0, K); //Send sample key to Plow
      Read(P0, Pi, binKeys); //Reads all  sample Keys from Plow
}

//Divide list into bins
MakeBins (bins, list, binKeys, numBins);

for (exchangeConfig = r; exchangeConfig>=ringConfig; exchangeConfig--)
{
      //Divide bins  into 2 groups
      DivideBins(bins, upperList, lowerList, numBins);

      //Find ID of neighbor on ring
      Pnext = NextNode(exchangeConfig);
```

```
      //Send/Read bins to/from Pnext & Merge
      if (Pi > Pnext)   //Pi holds the upper half of bins
      {  PAR
            SEND(Pi, Pnext, lowerList);
            READ(Pnext, Pi, nextUpperList);
         MergeBins(bins, upperList, nextUpperList); //Merges upper bins

      }
      else  //Pi holds the lower half of bins
      {  PAR
            SEND(Pi, Pnext, upperList);
            READ(Pnext, Pi, nextLowerList);

         MergeBins(bins, lowerList, nextLowerList); //Merges lower bins
      }
      numBins = numBins /2;
  }//end for
}
```

For simplicity, Figure 9.8 - Figure 9.12 illustrate the steps for bin-collecting using 4 nodes. Originally each node has 4 keys -- P_0 has 4, 7, 8 and 11, P_1 has 3, 10, 21 and 31, P_2 has with 1, 15, 16 and 18 and P_3 has with 12, 22, 25 and 28. With bin-collecting, each node will be responsible for maintaining the order of a specified range of keys.

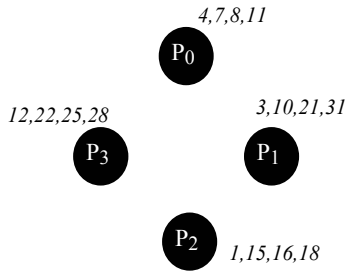


Figure 9.8: Bin-Collecting [Step 1 of 5]: Each node contains sorted list of 4 keys.

To determine the range for each node, each node sends the median key in its list to P_0. P_0 selects `7`, P_1 selects `10`, P_2 selects `15` and P_3 selects `22` as the median key. P_0 determines the range for each bin, and broadcasts this out to all nodes. In Figure 9.9, P_0 collects the sample keys, then sorts and broadcasts them to all nodes. The range for each bin, `0 - 3`, is given in Table 9.1.

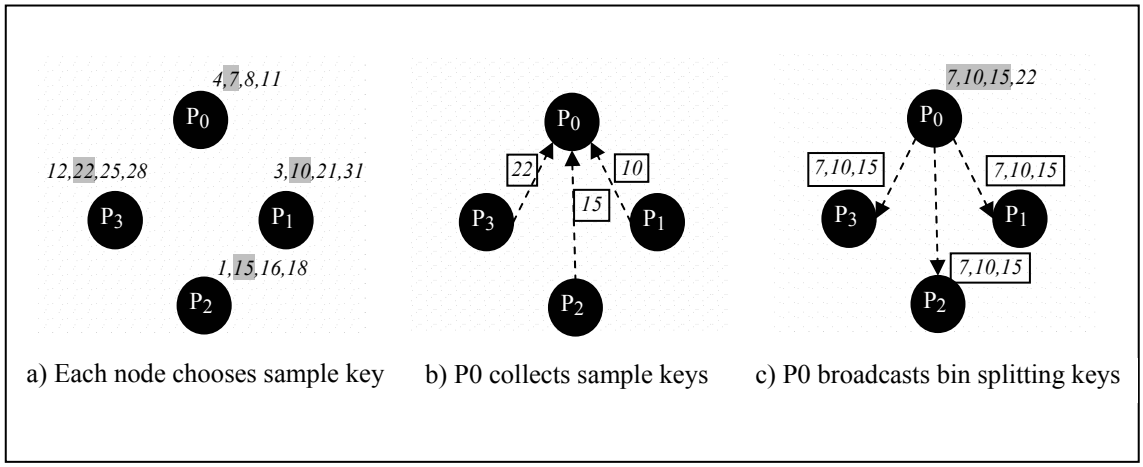


Figure 9.9: Bin-Collecting [Step 2 of 5]: Distributing splitting keys.

Table 9.1: Range of keys for each node using bin sorting

Bin	Range of Keys	Actual Keys
0	<=7	1, 3, 4, 7
1	> 7 and <= 10	8, 10
2	> 10 and <= 15	11, 12, 15
3	> 15	16, 18, 21, 22, 25, 28, 31

In Figure 9.10, each node has separated its keys into different bins. Some bins may be empty. For example, there are no keys in P_0's bin 3, P_1's bin 2 and P_3's bins 0 and 1.

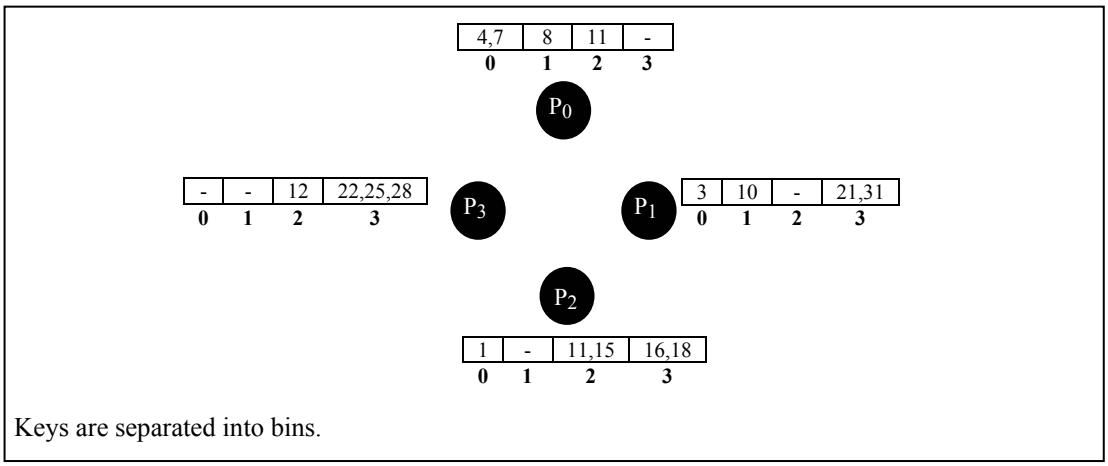


Figure 9.10: Bin-Collecting[Step 3 of 5]: Each list is divided into bins.

In Figure 9.11, P_0 and P_1 send the upper bins and P_2 and P_3 send the lower bins to their neighbor. The received bins are merged with the local bins and the bins that are sent are deleted from each node.

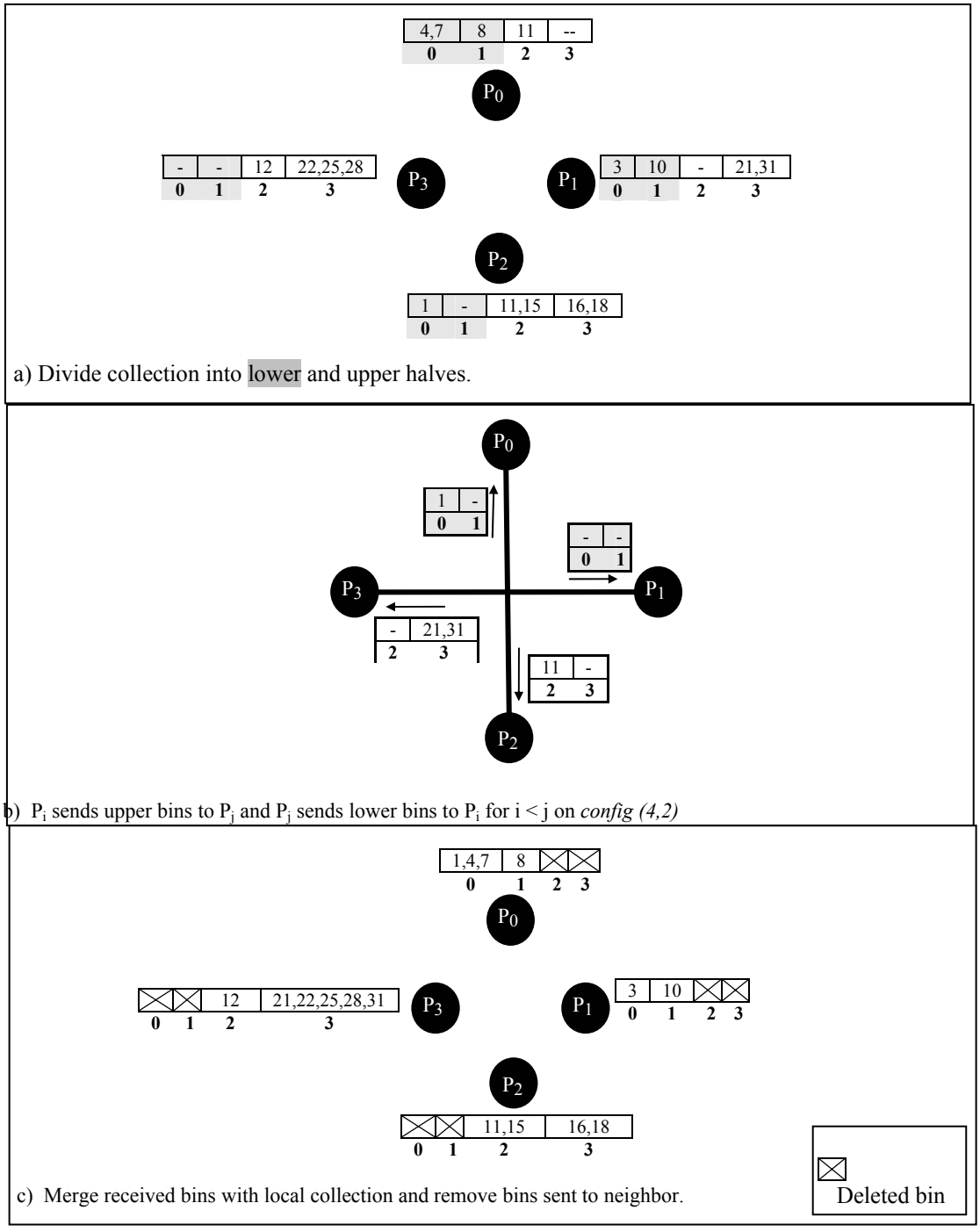


Figure 9.11: Bin-Collecting [Step 4 of 5]: Merging bins with neighbor on *config(4,2)*.

In Figure 9.12, in *config(4,1)* the nodes connect to different neighbors. As always P_i sends upper bins to P_j for $i < j$. P_0 and P_2 send their upper bin and P_1 and P_3 send their lower bin to their neighbor. The received bins are merged with the local collection and the sent bins are removed. This is the last step – the keys are now in sorted order.

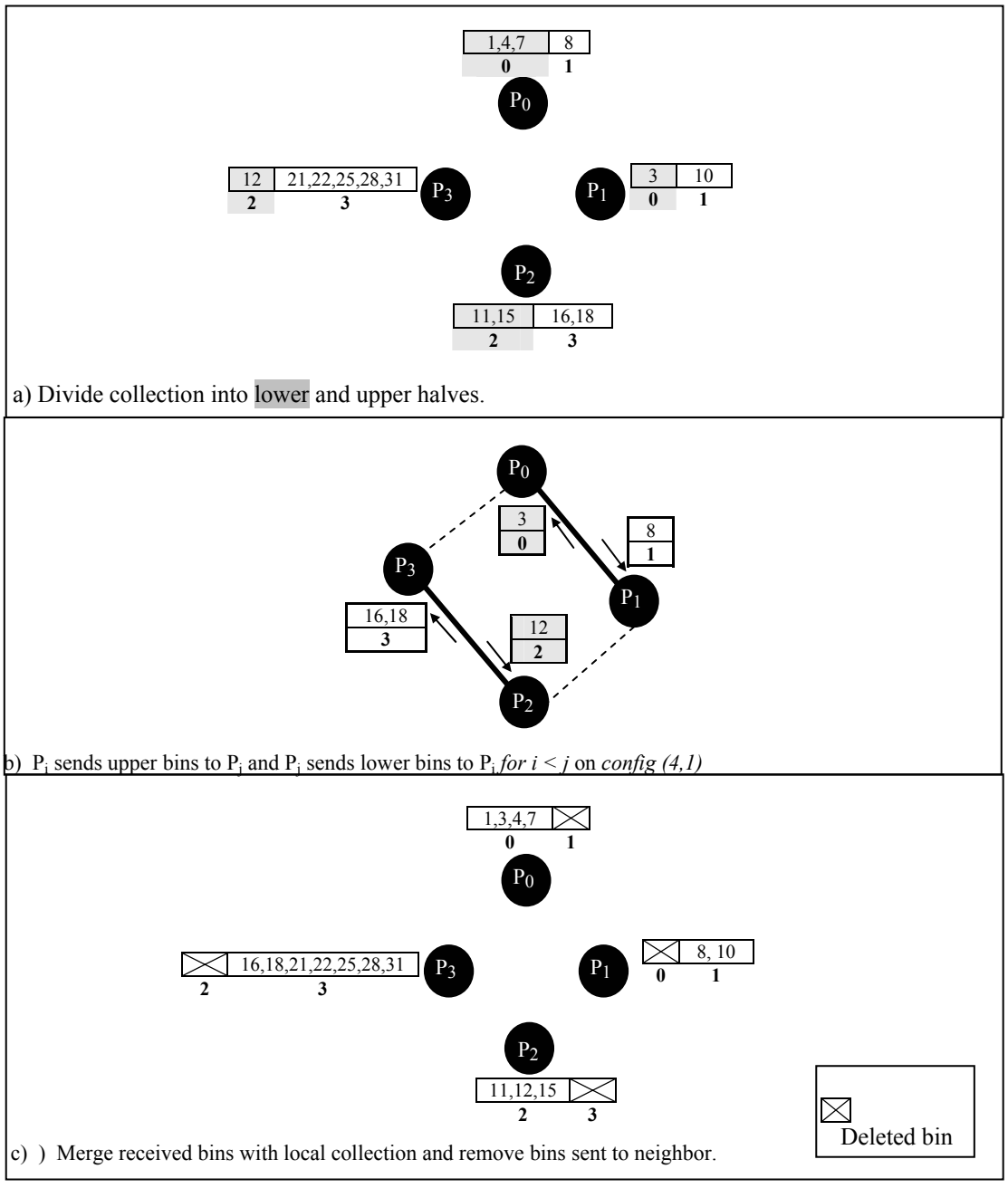


Figure 9.12: Bin-Collecting [Step 5 of 5]: Merging bins with neighbor on *config(4,1)*.

9.4 Analysis of algorithms

In the following analyses, t_c represents the unit of computational cost, t_e represents the unit cost to simultaneously exchange data (send and receive operation), and t_s represents the unit cost required to switch configurations on the MultiRing.

The expected case of sorting *M* keys on one node with quicksort is $O(M \log M)\ t_c$. In the parallel algorithms described in this chapter, instead of having one node sort *M* keys, the keys are divided evenly among *N* nodes. Each node receives *M/N* keys. The expected case of sorting *M/N* keys on one node with quicksort is $O(M/N \log M/N)\ t_c$.

When implementing parallel algorithms on the MultiRing, not only the computational cost for sorting the original sub-list of keys, but also the computational costs for merging the sub-lists and the cost to exchange data and reconfigure the switch during the collecting phase must be considered. The following analysis of the bitonic, MultiQuicksort and bin-collecting algorithms for the MultiRing are similar to their implementation for the hypercube; however the additional cost to reconfigure the MultiRing, t_s, is added.

With *M* keys and $N = 2^r$ processors, the bitonic sort has the following expected cost: $O\left(\left(\frac{M}{N}\log\frac{M}{N}\right)t_c + \left(\frac{r(r+1)}{2}\right)\left(\frac{M}{N}\right)(t_c + t_e) + r(t_s)\right)$. It takes $O\left(\left(\frac{M}{N}\log\frac{M}{N}\right)t_c\right)$ for the local quicksort on each node, $O\left(\left(\frac{r(r+1)}{2}\right)\left(\frac{M}{N}\right)(t_c + t_e)\right)$ for merging the lists and $O(r(t_s))$ for reconfiguring the MultiRing. Merging takes longer and longer as the lists grow in size.

The MultiQuicksort has the expected cost of :

$$O\left(\left(\frac{M}{N}\log\frac{M}{N}\right)t_c + r\log\left(\frac{M}{N}\right)t_c + r\left(\frac{M}{N}\right)(t_c) + \left(\frac{r(r+1)}{2}\right)(t_e + t_s) + r\left(\frac{M}{2N}\right)(t_e) + r(t_s)\right)$$

where $O\left(\left(\frac{M}{N}\log\frac{M}{N}\right)t_c\right)$ is the cost for local quicksort, $O\left(r\log\left(\frac{M}{N}\right)t_c\right)$ is the cost of using a binary search to find the key to separate the lists into two bins, $O\left(r\left(\frac{M}{N}\right)t_c\right)$ is the cost of merging subsequences, $O\left(\left(\frac{r(r+1)}{2}\right)(t_e+t_s)\right)$ is the cost of broadcasting medians to nodes in logical group, $O\left(r\left(\frac{M}{2N}\right)(t_e)\right)$ is the cost of exchanging subsequences and $O(r(t_s))$ is the cost of reconfiguring the network in order to exchange subsequences.

The bin collecting algorithm has the expected cost of

$$O\left(\left(\frac{M}{N}\log\frac{M}{N}\right)t_c+(t_c)+2\left(\frac{r(r+1)}{2}\right)(t_e+t_s)+(N\log(N))t_c+\left((N-1)\log\frac{M}{N}\right)t_c+r\left(\frac{M}{2N}\right)(t_e)+(r\,t_s)\right)$$

where $O\left(\left(\frac{M}{N}\log\frac{M}{N}\right)t_c\right)$ is the cost for local quicksort, $O(t_c)$ is the cost for each node to select median key, $O\left(2\left(\frac{r(r+1)}{2}\right)(t_e+t_s)\right)$ is the cost of P_0 collecting all sample keys and broadcasting them to the other nodes, $O((N\log(N))t_c)$ is the cost of sorting the samples, $O\left(\left((N-1)\log\frac{M}{N}\right)t_c\right)$ is the cost of each node splitting its keys into N bins, $O\left(r\left(\frac{M}{2N}\right)(t_e)\right)$ is the cost of exchanging subsequences, and $O(r\,t_s)$ is the cost of reconfiguring the network in order to exchange subsequences.

9.5 Concluding Remarks

In this chapter three parallel sorting algorithms were described for implementation on a MultiRing. The code for these algorithms have been included in the Appendix.

CHAPTER 10 : CONCLUSION AND FUTURE WORK

The MultiRing can be used in any area that requires parallel and distributed processing. Algorithms that could benefit from parallel processing are those that have data and/or function decomposition. Data decomposition involves partitioning data across several processors and applying the same function to the data. Function decomposition involves dividing the overall task into sub-tasks where each processor is responsible for executing one sub-task. (Pipelining is one example of how function decomposition enables parallel processing.) Whether data or function decomposition is used, the MultiRing can be used to connect processors for communication [8].

Industrial, medical, military and scientific applications that process large volumes of data in "real-time" are all target areas for using the MultiRing network. Designing automotive and aeronautical parts, analyzing mammograms for cancerous cells, computing battlefield statistics to plan invasions and modeling regional and global terrestrial water and energy cycles are all viable applications for use on the MultiRing. Specifically Arabnia and Bhandarkar [8-11,13-17] have developed algorithms for a real-time computer vision system.

10.1 Advantages of MultiRing Over Current Networks

The main issue between static and dynamic networks is the choice of increasing communication ports or switches. A high node degree in static networks can lead to complex multiplexing strategies for the numerous ports for each node and also to high rewiring expansion

costs. In a dynamic network, performance is hindered by the fall-through latency;needed to navigate through the collection of switches.

The hypercube networks were initially popular because parallel algorithms designed for other network topologies could be implemented on the hypercube without many changes. However, expanding the hypercube to allow additional processors requires changing the node degree and this makes manufacturing hypercubes complicated.

In an ever-changing environment, a parallel system must be able to expand easily so that it can accommodate additional processors while maintaining efficient communication between processors. The MultiRing is designed to meet these two needs. The MultiRing is scaleable since all nodes connect to a MultiRing switch with only two links. Inside the MultiRing switch is a dynamic interconnection network of switching elements that grows as processors are added.

To reduce conflicts most multi-stage interconnection networks employ a distributed control strategy where smart switches are employed to determine a message's communication path by its destination address. These smart switches are more expensive than simple switches. The MultiRing is an inexpensive network to build, it requires simple switches that are globally controlled. The communication paths are limited from the onset since only certain reconfigurations of the switch are allowed. These configurations form multiple rings which allow pairs of processors to communicate simultaneously.

Reconfigurable buses and meshes are topics of interest in network design. However, there are many other reconfigurable network proposed; each tends to serve a particular domain of applications. The MultiRing should support a large number of such domains. Reconfigurable meshes are characterized by unpredictable signal delays due to long path and number of processors connected to the wire carrying the signal [22-23]. In some cases " clock timings are

based on the worst-case shortest path, which degrades performance overall for the system." The MultiRing is based on one of the simplest and cost-effective networks; a ring. Only two links are needed to connect each processor to a MultiRing switch regardless of the number of processors. 2-D reconfigurable meshes require four links per processor and as the dimension increases, more links are needed. The MultiRing switch is considered a dumb switch and is modularly constructed from inexpensive switching elements. The reconfigurable mesh could require more complicated switches to restrict access to a common bus and monitor for concurrent writes.

The MultiRing network supports a heterogeneous environment; it does not required the same types of processors to be used. A virtual multi-processor can be created using existing local networks of workstations. Once synchronous communication is established, high performance applications can execute on the MultiRing Network as if a single parallel computer is used.

Cost is one reason why multiprocessors are not being used. MultiRings are a low cost way of building a virtual multiprocessor from existing technology.

10.2 Contributions of this Research

The goal of this dissertation was to enhance the existing MultiRing network proposed by Arabnia [8]. An altogether different approach to communication on the MultiRing was taken than what had previously been published. New communication strategies based on the premise that rings of nodes can work together to solve a problem independently from other rings were presented. Now, communication is restricted to involve nodes only on the same ring. At the lower level of programming, new communication strategies for simulating static networks are presented.

The definition of MultiRing is altered slightly in that the original design of the MultiRing included radial links that provided for a pipeline of rings [8], and these radial links have been eliminated. A new scaleable digital layout that supports easy expansion has been developed. The MultiRing switch's redesigned makes it possible for it to expand to allow additional processors with minimal cost – without disrupting the entire network. Alternatives for creating unidirectional and bidirectional MultiRing Switches that support automatic reconfiguration are also provided.

A MultiRing simulator has been created on which programmers can create and test parallel algorithms. An entire chapter of this dissertation is to collections of sorting algorithms for that have been implemented on the MultiRing simulator.

The MultiRing Switch proposed in this dissertation is a globally controlled multi-stage interconnection network. The switching elements in multi-stage interconnection networks that route messages by stripping the first bit from the message header are more complex than the switching elements used in our switch design. The logic components used in most reconfigurable mesh switches in most cases involves both global and local control; these components are more complex than the switching elements proposed for the MultiRing. In practice "simple" translates to "cheap." The MultiRing can be built more cheaply from simple switching elements than if complex switching elements were used.

The trade-offs to using simple switching elements with global control include: 1) all nodes must know the current configuration before initiating a send; 2) a delay is introduce to ensure the signal of the control bit has propagated up to all switching elements; 3) there cannot be any "partial" configurations of the switch where a subset of nodes are configured differently;

and 4) dynamic reconfigurations are not possible – no messages must be passing through the switch when it reconfigures. However, these are issues in other parallel architectures.

10.3 Future Work

Fault tolerance was not a part of this dissertation; however in the case where a single link fails (not the entire switch), adaptive routing would be useful. With adaptive routing, a node can choose an alternate path if the initial path is blocked Adaptive routing is not widely used in current parallel machines [26]. The nCube/3 had minimal adaptive routing in the hypercube, and a few others …CRAY T3 and Tera machine use adaptive routing. In general, adaptive routing adds to the complexity of the switch which will make the switch slower. Culler et al [26] also state that "fault tolerance and channel utilization can be obtained with a limited degree of adaptive routing." Therefore, sending data on the right link instead of the left link or waiting until the next configuration may be all that is needed for fault tolerance on the MultiRing. This is a topic for future work.

Optical pipelined buses are gaining in popularity [44, 52, 60, 66]. The optical buses have unidirectional signal propagation with a predictable delay which allow synchronized concurrent access in a pipelined fashion. Future work includes exploring how to incorporate optical technology with the MultiRing.

In this dissertation, machines are dedicated to running one problem. Additional research is needed to study multitasking on the MultiRing. For example, what happens when 4 processors are needed but only 2 are available?

Adding flow control policies to increase the performance of the MultiRing should also be explored. Two options include 1) setting the time spent on each configuration by the length of each message and/or the number of nodes requiring each configuration and 2) organizing

internal buffers to group independent messages that should be transmitted on the same configuration into larger composite messages.

The overall vision of the MuliRing network is a system that allows application programmers to execute existing high-performance applications in real-time on a collection of locally connected heterogeneous processors. This network would be a low cost network that could able to adapt to an ever-changing environment and easily accommodate additional processors.

REFERENCES

[1] V. Annamalai, C. S. Krishnamoorthy and V.Kamakoti. Adaptive finite element analysis on a parallel and distributed environment. *Parallel Computing*, Vol. 25, No. 12, November 1999. pp. 1413-1434.

[2] Gita Alaghband. Home Page. <http://carbon.cudenver.edu/~galaghba/gita.html>.

[3] A. John Anderson. *Multiple Processing A Systems Overview.* Prentice Hall, 1989.

[4] H. R. Arabnia and M .A. Oliver. A reconfigurable network of processors. *The International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '95)*. pp. 134-148. 1995.

[5] H. R. Arabnia. and M. R Robinson. Parallelizing using process-and-data decomposition (PADD) Approach on a Multi-Ring Transputer Network - An Example. *Transputer Research and Applications - NATUG* 3, pp. 107-118. 1990

[6] H. R. Arabnia. Real-time preprocessing of multiple images using an unconventional pipeline approach. *Proceedings of the 4th Conference of North American Transputer Users Group (NATUG 4)*, pp. 57-66. 1990.

[7] Hamid R. Arabnia. Two parallel sort algorithms: Pipeline-sort and MultiRing sort. *Proceedings of the Fifth North American Transputers, Users Group Conference,* pp. 1-11, 1992.

[8] Hamid R. Arabnia. A transputer-based reconfigurable parallel system. *Proc.*

Transputer Research and Applications –NATUG 6, pp. 153-167, 1993.

[9] H. R. Arabnia Broadcasting mechanisms on the reconfigurable multiring network. *Proceedings of the 14th IMACS World Congress on Computational and Applied Mathematics*, vol. 3 pp. 1076-1080. 1994.

[10] H. R. Arabnia. Fast fourier transform on the reconfigurable multiring network. *Proceedings of the 14th IMACS World Congress on Computational and Applied Mathematics,* vol. 3 pp. 1080-1084. 1994.

[11] Hamid R. Arabnia. Stereo correspondence problem on a ring-base network. *Aizu International Symposium on Parallel Algorithms/Architecture Synthesis.* IEEE, Piscataway, NJ. 1997, pp. 265-275.

[12] Hamid R. Arabnia and Jeffery W. Smith. A reconfigurable interconnection network for imaging operations and its implementation using a multi-stage switching box. *Proc. 1993 Conf. High Performance Computing: New Horizons,* pp.: 349-357, Alberta, Canada 1993

[13] S. M. Bhandarkar and H. R. Arabnia. The multi-ring reconfigurable multiprocessor network for computer vision. *Proceedings IEEE International Workshop Computer Architectures for Machine Perception (CAMP 93)* pp. 180-190, New Orleans, Dec 15-17, 1993.

[14] S. M. Bhandarkar and Arabnia H. R., Parallel 3-d object recognition. *Proceedings of The International Conference on Signal Processing (ICSP'93)*, Beijing, China, pp. 1022 - 1026. 1993

[15] Suchendra M Bhandarkar and Hamid R. Arabnia. REFINE multiprocessor - theoretical properties and algorithms. *Parallel Computing*, Vol. 21, No. 11, pp.

1783-1805, 1995

[16] S. M. Bhandarkar and H. R. Arabnia. The hough transform on a reconfigurable multi-ring network. *Journal of Parallel and Distributed Computing*, vol. 24, No. 1, January, pp. 107-114. 1995.

[17] Suchendra M Bhandarkar and Hamid R Arabnia. Parallel computer vision on a reconfigurable multiprocessor network. *IEEE Transactions on Parallel and Distributed Systems*, Vol 8, No 3, pp. 292-309, 1997

[18] S. M. Bhandarkar, H. R. Arabnia and J. W. Smith, A reconfiguratble Architecture for Image Processing and Computer Vision. *International Journal of Pattern Recognition and Artificial Intelligence, vol. 9, no. 2, pp. 2001-229, 1995*

[19] A. Bouridane, D. Crookes, P. Donachy, K. Alotaibi and K. Benkrid. A high level FPGA-based abstract machine for image processing, *Journal of Systems Architecture,* Volume 45, Issue 10, April 1999, pp. 809-824.

[20] Domingo Benitez. Performance of reconfigurable architectures for image-processing applications, *Journal of Systems Architecture*, Volume 49, Issues 4-6, September 2003, pp. 193-210

[21] A. Bond and D. Fashena. Parallel vision techniques on the hypercube computer, G. Fox editor, *The Third Conference on Hypercube Concurrent Computers and Applications,* ACM, 1998 Vol. 2, pp. 1007-1010

[22] Kiran Bondalapati and Viktor K. Prasanna. Reconfigurable Computing: Architectures, Models and Algorithms. *Current Science*, vol. 78, no. 7, pp. 828--837, April 2000.

[23] Kiran Bondalapati and Viktor K. Prasanna. Reconfigurable Meshes: Theory and Practice. *Reconfigurable Architectures Workshop, International Parallel Processing Symposium,* April 1997.

[24] Prashanth B. Bhat, Viktor K. Prasanna and C. S. Raghavendra. Adaptive communication algorithms for distributed heterogeneous systems. *Journal of Parallel and Distributed Computing,* Vol. 59. No 2. November 1999, pp. 252-279.

[25] G. M. Chaudhry and N. Khan A. Bandwidth of a reconfigurable multiple-group multiprocessor system, *Journal of Systems Architecture,* Volume 42, Issue 3, 1 October 1996, pp. 225-234.

[26] David E. Culler and Jaswinder Pal Sing, with Anoop Gupta. *Parallel Computer Architecture: Hardware/Software Approach*. Morgan Kaufman Publishers Inc, 1999.

[27] *Science Team III Grand Challenge Investigators for Computational Technologies for Earth and Space Sciences (COMTESS)*. 2003. <http://sdcd.gsfc.nasa.gov/ESS/grand.st3.html>

[28] Thomas L. Casavant, Pavel Tvrdik and Frantisek Plasil. *Parallel Computer: Theory and Practice*. IEEE Computer Society Press, 1996.

[29] José Alberto Fernández-Zepeda, Ramachandran Vaidyanathan and Jerry L. Trahan. Using Bus Linearization to Scale the Reconfigurable Mesh. *Journal of Parallel and Distributed Computing,* Volume 62, Issue 4, April 2002, pp. 495-516.

[30] Hamid R. Arabnia and Xiangjian He. Scalable switch for uni-directional

MultiRing network.[pending]

[31] Xiangjian He and Hamid R. Arabnia. Scalable switch for bi-directional MultiRing network. [pending]

[32] R. W. Hockney and C. R. Jesshope. *Parallel Computers*. Adam Hilger Ltd, 1981.

[33] *HPCC Grand Challenges*. <http://www.nhse.org/grand_challenge.html>

[34] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[35] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing*. McGraw-Hill, 1998.

[36] N.J.A. Sloane. *On-Line Encyclopedia of Integer Sequences*. 2003 <http://www.research.att.com/~njas/sequences/>

[37] Harry Jordan and Gita Alaghband. *Fundamentals of Parallel Processing*. Prentice Hall, 2003.

[38] Ju-wook Jang, Heonchul Park, and Viktor K. Prasanna. An Optimal Multiplication Algorithm on Reconfigurable Mesh. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 5, May 1997. pp. 521-532.

[39] top500@rz.uni-mannheim.de. *The Kendall Square Research KRS1*. 1995. <http://www.netlib.org/benchmark/top500/reports/report94/Archite/node22.html>

[40] Amitava Majumdar. *Parallelization on the Kendal Square KSR-1*. 1999. <http://www.sdsc.edu/~majumdar/thesis/nodes52.html>

[41] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes* M. Kaufmann Publishers, 1992.

[42] Patrick Lidstone, Michael Horwood and Jim Baker. A dynamically reconfigurable parallel processing system, *Microprocessors and Microsystems,*

Volume 19, Issue 3, 1995, pp. 157-160.

[43] R. Lin, S. Olariu, J.L. Schwing and J. Zhang. Sorting in O(1) time on a reconfigurable mesh of size nxn. *Parallel Computing: From Theory to Sound Practice, Proceedings of EWPC'92*, Plenary Address, IOS Press, 16-27, 1992

[44] Kequin Li, Yin Pan and Mounir Hamdi. Solving graph theory problems using a reconfigurable pipeline optical buses. *Parallel Computing*, Vol. 26, No. 6. May 2000. pp. 723-735.

[45] Thomas J. LeBlanc, Michael L. Scott and Christopher M. Brown. Large-scale parallel programming: experience with the BBN Butterfly parallel processor. *Proceedings of the ACM/SIGPLAN PPEALS 1988*, pp. 161-172, 1998

[46] Todd C. Mowry. *Multiprocessor Interconnection Network.* CS Department at Carnegie Mellon University <http://www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15740-f98/public/lectures/mp_networks.pdf>

[47] Russ Miller, V.K. Prasanna-Kumar, Dionisios I. Reisis and Quentin F. Stout. Parallel Computations on Reconfigurable Meshes. *IEEE Transactions on Computers* 42 (1993), pp. 678-692.

[48] Bruce M. Maggs and Ramesh K. Sitaraman. Simple algorithms for routing on butterfly networks with bounded queues. *SIAM Journal of Computing*, Vol 28, No 3, pp. 984-1003, 1999.

[49] Stephan Olariu and James L. Schwing. A New Deterministic Sampling Scheme with Applications to Broadcast-Efficient Sorting on the Reconfigurable Mesh. *Journal of Parallel and Distributed Computing*, 32 (1996) pp. 215-222.

[50] An optimal solution for ATM switches. *Computer Networks and ISDN Systems,* Vol. 29. No. 17-18, February 1998, pp. 2039-2052.

[51] Sanguthevar Rajasekaran, Wang Chen, and Shibu Yooseph. Unifying themes for selection on any network. *Journal of Parallel and Distributed Computing.* (46) 1997 pp. 105-111

[52] Sibabrata Ray and Hong Jiang. A reconfigurable bus structure for multiprocessors with bandwidth reuse, *Journal of Systems Architecture,* Volume 45, Issue 11, May 1999, pp. 847-862.

[53] Sanguthevar Rajasekaran and Theodore McKendall. Permutation Routing and Sorting on the Reconfigurable Mesh. *International Journal of Foundations of Computer Science* 9(2), 1998, pp. 199-211.

[54] Seyed H. Roosta. *Parallel Processing and Parallel Algorithms: Theory and Computation*. Springer, 2000.

[55] Steven Seidel and William George. Binsorting on hypercubes with d-port communication, G. Fox Editor, *The Third Conference on Hypercube Concurrent Computers and Applications,* ACM, 1998 Vol. 2, pp. 1455-1461

[56] Harold S. Stone. *High-Performance Computer Architecture*. 3rd ed. Addison-Wesley, 1993.

[57] Q. F. Stout and B. Wagar. Passing message in link-bound hypercubes, *Hypercube Multiprocessors 1987*, SIAM Press, Phila., ,pp. 251-257. 1987

[58] Steven R. Seidel and Lynn R. Ziegler. Sorting Hypercubes, *Hypercube Multiprocessors 1987*, SIAM Press, Phila., pp. 285-291. 1987

[59] Thiab R. Taha and Hamid R. Arabnia. Exploiting a ring-based MIMD

multicomputer for numerical problems. *PROC 1993 IEEE Region 10 Conference on Computer, Communication, Control and Power Engineering (TENCON '93),* IEEE, PISCATAWAY, NJ, (USA), 1993, pp. 221-224.

[60] Jerry L. Trahan, Anu G. Bourgeois, Yi Pan and Ramachandran Vaidyanathan.Optimally Scaling Permutation Routing on Reconfigurable Linear Arrays with Optical Buses. *Journal of Parallel and Distributed Computing,* Volume 60, Issue 9, September 2000, pp. 1125-1136.

[61] Horng-Ren Tsai, Shi-Jinn Horng, Shun-Shan Tsai, Shung-Shing Lee, Tzong-Wann Kao and Chia-Ho Chen. Optimal Speed-Up Parallel Image Template Matching Algorithms on Processor Arrays with a Reconfigurable Bus System,', *Computer Vision and Image Understanding,* Volume 71, Issue 3, September 1998, pp. 393-412.

[62] *TOP500 Supercomputer sites*. <http://www.top500.org>

[63] Jerry L. Trahan , Ramachandran Vaidyanathan and Ratnapuri K. Thiruchelvan. On the Power of Segmenting and Fusing Buses, *Journal of Parallel and Distributed Computing*, Volume 34, Issue 1, 10 April 1996, pp. 82-94.

[64] B. Wagar, Hyperquicksort: a fast sorting algorithms for hypercubes, *Hypercube Multiprocessors 1987*, SIAM Press, Phila., pp. 292-299, 1987

[65] C. H. Wu, S. J. Horng and H. R. Tsai. Efficient Parallel Algorithms for Hierarchical Clustering on Arrays with Reconfigurable Optical Buses. *Journal of Parallel and Distributed Computing.* Vol. 60, No. 9, Sept. 2000, pp. 1137-1153.

[66] Chin-Hsiuing Wu, Shi-Jinn Horng and Horng-Ren Tsai. Optimal Parallel Algorithms for Computer Vision Problems, *Journal of Parallel and Distributed*

Computing, Vol. 62, No. 6. June 2002. pp. 1021-1041.

[67] Tiffani L. Williams. *CS442/EECE443 Introduction to Parallel Computing 2. Interconnection Networks*. 2002. <http://www.cs.unm.edu/~tlw/cs442>

[68] Vlad Wojcik. *COSC 3P93: Parallel Computing.* Brock University Department of Computer Science. <http://www.cosc.brocku.ca/Offerings/3P93/>

APPENDIX A:DEVELOPING ROUTING ALGORITHMS

Several different routing algorithms were created during the writing of this dissertation. The initial goal was to create an arithmetic (algorithmic) routing method that nodes could use to decide the output channel when a message arrived based on calculations involving current position and destination of the message. This method would then be used by each node during initialization of the MultiRing network to create a routing table that enabled table driven routing. The table would contain the next 'step' for a message. As a result both arithmetic and table driven routing would be supported. Appendix A describes the steps taken to develop five different routing algorithms for the MultiRing.

(AppedixA.pdf)

APPENDIX B : SORTING CODE

B.1 Bitonic Sort (bit.c)

B.2 MultiQuickSort (mquick.c)

B.3 BinSort (binsort.c)

B.4 sortHelp.h

APPENDIX C : MULTIRING CODE